FIG. 1

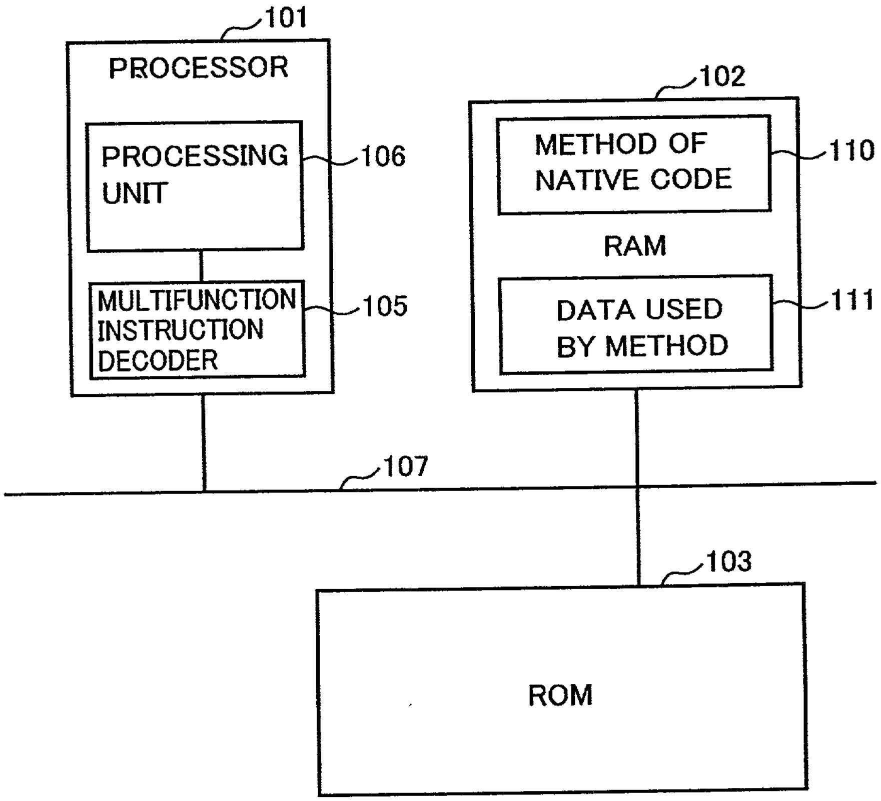

101
PROCESSOR
PROCESSING UNIT
106
MULTIFUNCTION INSTRUCTION DECODER
105
102
METHOD OF NATIVE CODE
110
RAM
DATA USED BY METHOD
111
107
103
ROM

FIG. 2

MNEMONIC		OPERATION	CONDITION BIT (C)
ADD	Rdest,Rsrc	Rdest = Rdest + Rsrc	—
ADD3	Rdest,Rsrc,#imm16	Rdest = Rsrc + (sh)imm16	—
ADDI	Rdest,#imm8	Rdest = Rdest + (sb)imm8	—
ADDV	Rdest,Rsrc	Rdest = Rdest + Rsrc	CHANGE
ADDV3	Rdest,Rsrc,#imm16	Rdest = Rsrc + (sh)imm16	CHANGE
ADDX	Rdest,Rsrc	Rdest = Rdest + Rsrc + C	CHANGE
AND	Rdest,Rsrc	Rdest = Rdest & Rsrc	—
AND3	Rdest,Rsrc,#imm16	Rdest = Rsrc & (uh)imm16	—
BC	pcdisp8	if(C) PC=PC+((sb)pcdisp8<<2)	—
BC	pcdisp24	if(C) PC=PC+((s24)pcdisp24<<2)	—
BEQ	Rsrc1,Rsrc2,pcdisp16	if(Rsrc1 == Rsrc2) PC=PC+((sh)pcdisp16<<2)	—
BEQZ	Rsrc,pcdisp16	if(Rsrc == 0) PC=PC+((sh)pcdisp16<<2)	—
BGEZ	Rsrc,pcdisp16	if(Rsrc >= 0) PC=PC+((sh)pcdisp16<<2)	—
BGTZ	Rsrc,pcdisp16	if(Rsrc > 0) PC=PC+((sh)pcdisp16<<2)	—
BL	pcdisp8	R14=PC+4,PC=PC+((sb)pcdisp8<<2)	—
BL	pcdisp24	R14=PC+4,PC=PC+((s24)pcdisp24<<2)	—
BLEZ	Rsrc,pcdisp16	if(Rsrc <= 0) PC=PC+((sh)pcdisp16<<2)	—
BLTZ	Rsrc,pcdisp16	if(Rsrc < 0) PC=PC+((sh)pcdisp16<<2)	—
BNC	pcdisp8	if(!C) PC=PC+((sb)pcdisp8<<2)	—
BNC	pcdisp24	if(!C) PC=PC+((s24)pcdisp24<<2)	—
BNE	Rsrc1,Rsrc2,pcdisp16	if(Rsrc1 != Rsrc2) PC=PC+((sh)pcdisp16<<2)	—
BNEZ	Rsrc,pcdisp16	if(Rsrc != 0) PC=PC+((sh)pcdisp16<<2)	—
BRA	pcdisp8	PC=PC+((sb)pcdisp8<<2)	—
BRA	pcdisp24	PC=PC+((s24)pcdisp24<<2)	—
CMP	Rsrc1,Rsrc2	(s)Rsrc1 < (s)Rsrc2	CHANGE
CMPI	Rsrc,#imm16	(s)Rsrc < (sh)imm16	CHANGE
CMPU	Rsrc1,Rsrc2	(u)Rsrc1 < (u)Rsrc2	CHANGE
CMPUI	Rsrc,#imm16	(u)Rsrc < (u)((sh)imm16)	CHANGE
DIV	Rdest,Rsrc	Rdest = (s)Rdest / (s)Rsrc	—
DIVU	Rdest,Rsrc	Rdest = (u)Rdest / (u)Rsrc	—
JL	Rsrc	R14 = PC+4, PC = Rsrc	—
JMP	Rsrc	PC = Rsrc	—
LD	Rdest,@(disp16,Rsrc)	Rdest = *(s *)(Rsrc+(sh)disp16)	—
LD	Rdest,@Rsrc	Rdest = *(s *)Rsrc	—
LD	Rdest,@Rsrc+	Rdest = *(s *)Rsrc, Rsrc += 4	—

FIG. 3

MNEMONIC		OPERATION	CONDITION BIT (C)
LD24	Rdest,#imm24	Rdest = imm24 & 0x00ffffff	—
LDB	Rdest,@(disp16,Rsrc)	Rdest = *(sb *)(Rsrc+(sh)disp16)	—
LDB	Rdest,@Rsrc	Rdest = *(sb *)Rsrc	—
LDH	Rdest,@(disp16,Rsrc)	Rdest = *(sh *)(Rsrc+(sh)disp16)	—
LDH	Rdest,@Rsrc	Rdest = *(sh *)Rsrc	—
LDI	Rdest,#imm16	Rdest = (sh)imm16	—
LDI	Rdest,#imm8	Rdest = (sb)imm8	—
LDUB	Rdest,@(disp16,Rsrc)	Rdest = *(ub *)(Rsrc+(sh)disp16)	—
LDUB	Rdest,@Rsrc	Rdest = *(ub *)Rsrc	—
LDUH	Rdest,@(disp16,Rsrc)	Rdest = *(uh *)(Rsrc+(sh)disp16)	—
LDUH	Rdest,@Rsrc	Rdest = *(ub *)Rsrc	—
LOCK	Rdest,@Rsrc	LOCK = 1, Rdest = *(s *)Rsrc	—
MACHI	Rsrc1,Rsrc2	accumulator += (s)(Rsrc1 & 0xffff0000) * (s)((s)Rsrc2>>16)	—
MACLO	Rsrc1,Rsrc2	accumulator += (s)(Rsrc1<<16) * (sh)Rsrc2	—
MACWHI	Rsrc1,Rsrc2	accumulator += (s)Rsrc1 * (s)((s)Rsrc2>>16)	—
MACWLO	Rsrc1,Rsrc2	accumulator += (s)Rsrc1 * (sh)Rsrc2	—
MUL	Rdest,Rsrc	Rdest = (s)Rdest * (s)Rsrc	—
MULHI	Rsrc1,Rsrc2	accumulator = (s)(Rsrc1 & 0xffff0000) * (s)((s)Rsrc2>>16)	—
MULLO	Rsrc1,Rsrc2	accumulator = (s)(Rsrc1<<16) * (sh)Rsrc2	—
MULWHI	Rsrc1,Rsrc2	accumulator = (s)Rsrc1 * (s)((s)Rsrc2>>16)	—
MULWLO	Rsrc1,Rsrc2	accumulator = (s)Rsrc1 * (sh)Rsrc2	—
MV	Rdest,Rsrc	Rdest = Rsrc	—
MVFACHI	Rdest	Rdest = accumulater >> 32	—
MVFACLO	Rdest	Rdest = accumulator	—
MVFACMI	Rdest	Rdest = accumulator >> 16	—
MVFC	Rdest,CRsrc	Rdest = CRsrc	—
MVTACHI	Rsrc	accumulator[0:31] = Rsrc	—
MVTACLO	Rsrc	accumulator[32:63] = Rsrc	—
MVTC	Rsrc,CRdest	CRdest = Rsrc	CHANGE
NEG	Rdest,Rsrc	Rdest = 0 - Rsrc	—
NOP		/*no-operation*/	—
NOT	Rdest,Rsrc	Rdest = ~Rsrc	—
OR	Rdest,Rsrc	Rdest = Rdest | Rsrc	—
OR3	Rdest,Rsrc,#imm16	Rdest = Rsrc | (uh)imm16	—
RAC		Round the 32-bit value in the accumulator	—
RACH		Round the 16-bit value in the accumulator	—
REM	Rdest,Rsrc	Rdest = (s)Rdest % (s)Rsrc	—
REMU	Rdest,Rsrc	Rdest = (u)Rdest % (u)Rsrc	—
RTE		PC = BPC & 0xfffffffc, PSW[SM,IE,C] = PSW[BSM,BIE,BC]	CHANGE

FIG. 4

MNEMONIC	OPERATION	CONDITION BIT (C)
SETH Rdest,#imm16	Rdest = imm16 << 16	—
SLL Rdest,Rsrc	Rdest = Rdest << (Rsrc & 31)	—
SLL3 Rdest,Rsrc,#imm16	Rdest = Rsrc << (imm16 & 31)	—
SLLI Rdest,#imm5	Rdest = Rdest << imm5	—
SRA Rdest,Rsrc	Rdest = (s)Rdest >> (Rsrc & 31)	—
SRA3 Rdest,Rsrc,#imm16	Rdest = (s)Rsrc >> (imm16 & 31)	—
SRAI Rdest,#imm5	Rdest = (s)Rdest >> imm5	—
SRL Rdest,Rsrc	Rdest = (u)Rdest >> (Rsrc & 31)	—
SRL3 Rdest,Rsrc,#imm16	Rdest = (u)Rsrc >> (imm16 & 31)	—
SRLI Rdest,#imm5	Rdest = (u)Rdest >> imm5	—
ST Rsrc1,@(disp16,Rsrc2)	*(s *)(Rsrc2+(sh)disp16) = Rsrc1	—
ST Rsrc1,@+Rsrc2	Rsrc2 += 4, *(s *)Rsrc2 = Rsrc1	—
ST Rsrc1,@-Rsrc2	Rsrc2 -= 4, *(s *)Rsrc2 = Rsrc1	—
ST Rsrc1,@Rsrc2	*(s *)Rsrc2 = Rsrc1	—
STB Rsrc1,@(disp16,Rsrc2)	*(sb *)(Rsrc2+(sh)disp16) = Rsrc1	—
STB Rsrc1,@Rsrc2	*(sb *)Rsrc2 = Rsrc1	—
STH Rsrc1,@(disp16,Rsrc2)	*(sh *)(Rsrc2+(sh)disp16) = Rsrc1	—
STH Rsrc1,@Rsrc2	*(sh *)Rsrc2 = Rsrc1	—
SUB Rdest,Rsrc	Rdest = Rdest - Rsrc	—
SUBV Rdest,Rsrc	Rdest = Rdest - Rsrc	CHANGE
SUBX Rdest,Rsrc	Rdest = Rdest - Rsrc - C	CHANGE
TRAP #n	PSW[BSM,BIE,BC] = PSW[SM,IE,C] PSW[SM,IE,C] = PSW[SM,0,0] Call trap-handler number-n	CHANGE
UNLOCK Rsrc1,@Rsrc2	if(LOCK) { *(s *)Rsrc2 = Rsrc1; } LOCK=0	—
XOR Rdest,Rsrc	Rdest = Rdest ^ Rsrc	—
XOR3 Rdest,Rsrc,#imm16	Rdest = Rsrc ^ (uh)imm16	—

```
where:
typedef singed int      s;  /* 32 bit signed integer (word)*/
typedef unsigned int    u;  /* 32 bit unsigned integer (word)*/
typedef signed short    sh; /* 16 bit signed integer (halfword)*/
typedef unsigned short  uh; /* 16 bit unsigned integer (halfword)*/
typedef signed char     sb; /*  8 bit signed integer (byte)*/
typedef unsigned char   ub; /*  8 bit unsigned integer (byte)*/
```

FIG. 5

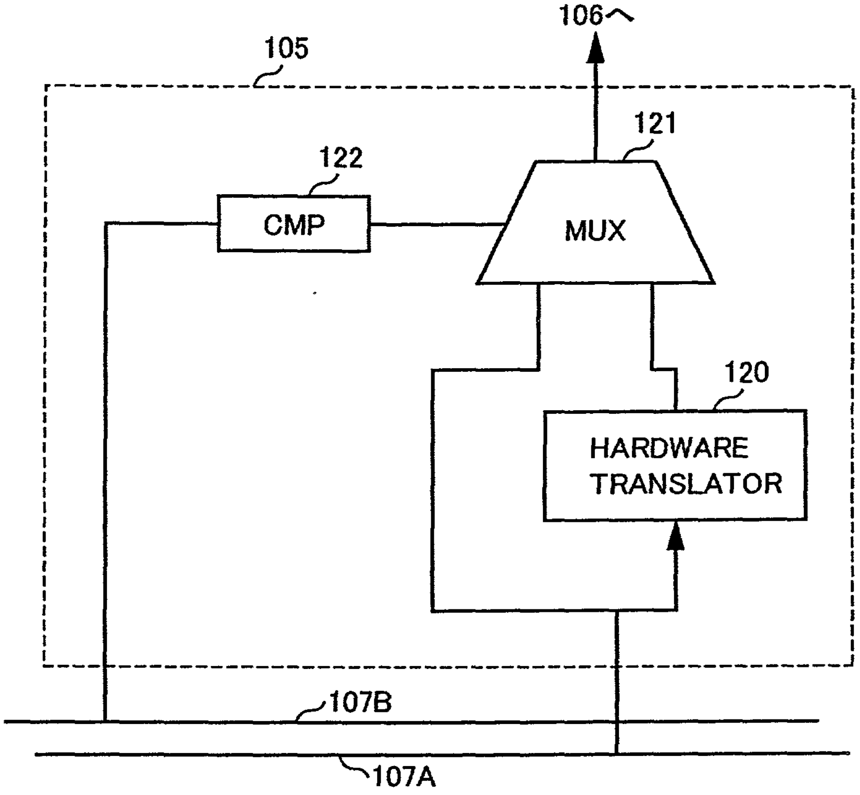


FIG. 6

ADDRESS	Java BYTECODE	MEANING
0	iload_0	PUSH LOCAL VARIABLE 0 ONTO STACK
1	iload_1	PUSH LOCAL VARIABLE 1 ONTO STACK
2	iadd	POP TWO INTEGERS FROM STACK TOP, ADD THEM, AND PUSH THE RESULT ONTO STACK
3	istore_2	POP INTEGER FROM STACK TOP, AND STORE IT INTO LOCAL VARIABLE 2
4	iconst_1	PUSH 1 ONTO STACK
5	iload_0	PUSH LOCAL VARIABLE 0 ONTO STACK
6	ifge 21	POP FROM STACK TOP, AND JUMP TO ADDRESS 21 IF POPPED VALUE IS EQUAL TO OR GREATER THAN 0
9	iconst_2	PUSH 2 ONTO STACK
10	iload_0	PUSH LOCAL VARIABLE 0 ONTO STACK
11	iload_1	PUSH LOCAL VARIABLE 1 ONTO STACK
12	iconst_3	PUSH 3 ONTO STACK
13	iload_2	PUSH LOCAL VARIABLE 2 ONTO STACK
14	iadd	POP TWO INTEGERS FROM STACK TOP, ADD THEM, AND PUSH THE RESULT ONTO STACK
15	idiv	POP TWO INTEGERS FROM STACK TOP, DIVIDE THE FIRST BY THE SECOND, AND PUSH THE RESULT ONTO THE STACK
16	iadd	POP TWO INTEGERS FROM STACK TOP, ADD THEM, AND PUSH THE RESULT ONTO STACK
17	imul	POP TWO INTEGERS FROM STACK TOP, MULTIPLY THEM, AND PUSH THE RESULT ONTO STACK
18	goto 28	JUMP TO ADDRESS 28
21	iload_0	PUSH LOCAL VARIABLE 0 ONTO STACK
22	iconst_1	PUSH 1 ONTO STACK
23	isub	POP TWO INTEGERS FROM STACK TOP, SUBTRACT THE SECOND FROM THE FIRST, AND PUSH THE RESULT ONTO STACK
24	iload_2	PUSH LOCAL VARIABLE 2 ONTO STACK
25	invokestatic <int F(int, int)>	CALL METHOD int F(int, int)
28	iadd	POP TWO INTEGERS FROM STACK TOP, AND PUSH ADDED RESULT ONTO STACK
29	ireturn	POP STACK TOP, AND JUMP TO SUBROUTINE CALLING SOURCE WITH POPPED VALUE AS RETURN VALUE

FIG. 7

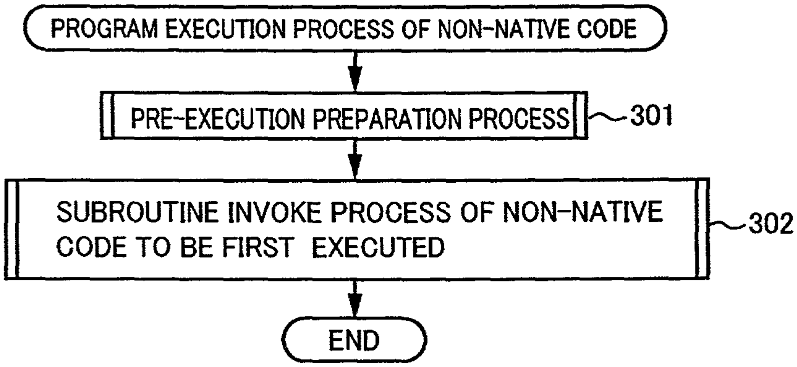
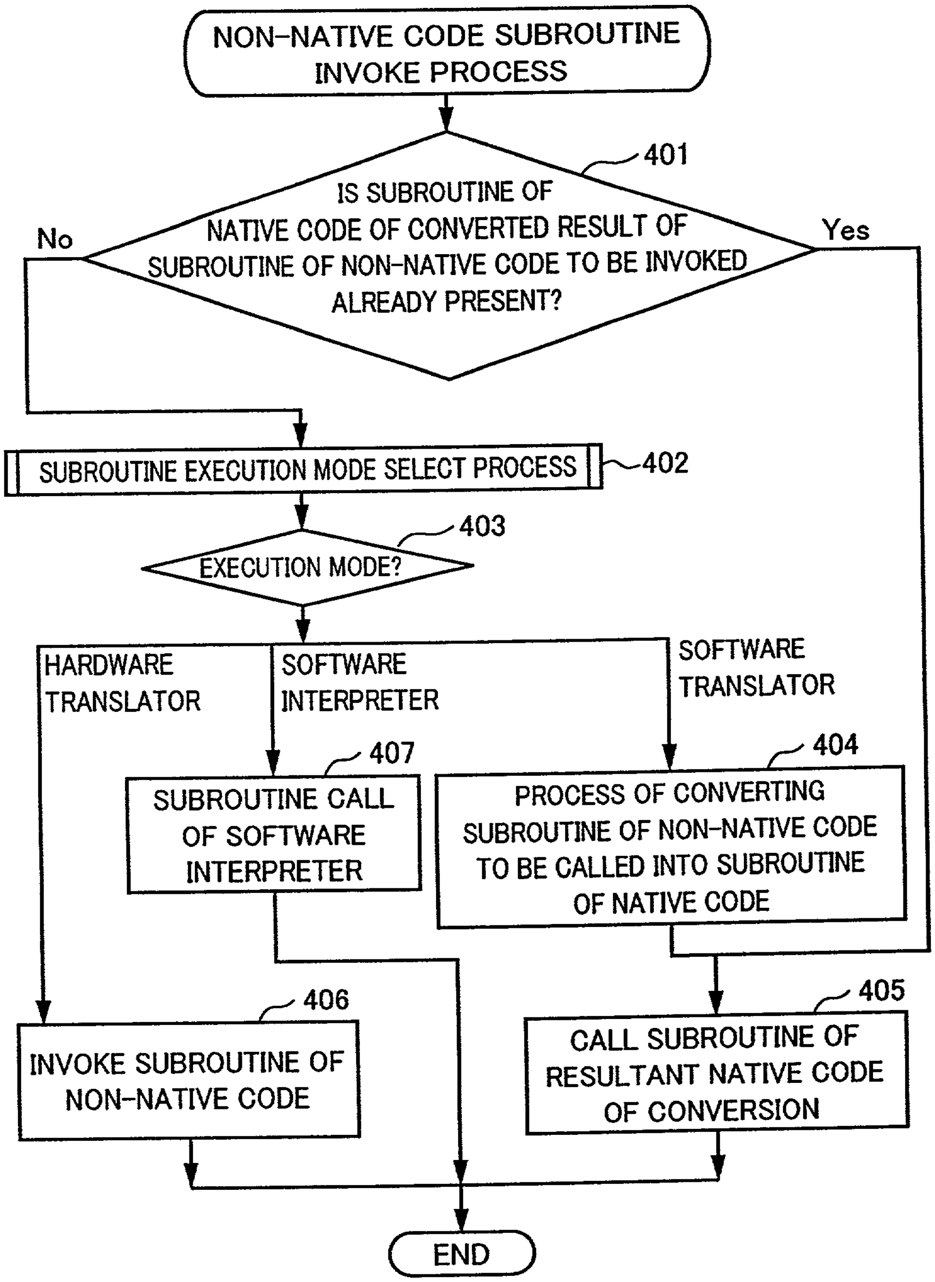

PROGRAM EXECUTION PROCESS OF NON-NATIVE CODE
PRE-EXECUTION PREPARATION PROCESS
301
SUBROUTINE INVOKE PROCESS OF NON-NATIVE CODE TO BE FIRST EXECUTED
302
END

FIG. 8



NON-NATIVE CODE SUBROUTINE INVOKE PROCESS
401
IS SUBROUTINE OF NATIVE CODE OF CONVERTED RESULT OF SUBROUTINE OF NON-NATIVE CODE TO BE INVOKED ALREADY PRESENT?
No
Yes
SUBROUTINE EXECUTION MODE SELECT PROCESS
402
403
EXECUTION MODE?
HARDWARE TRANSLATOR
SOFTWARE INTERPRETER
SOFTWARE TRANSLATOR
407
SUBROUTINE CALL OF SOFTWARE INTERPRETER
404
PROCESS OF CONVERTING SUBROUTINE OF NON-NATIVE CODE TO BE CALLED INTO SUBROUTINE OF NATIVE CODE
406
INVOKE SUBROUTINE OF NON-NATIVE CODE
405
CALL SUBROUTINE OF RESULTANT NATIVE CODE OF CONVERSION
END

FIG. 9

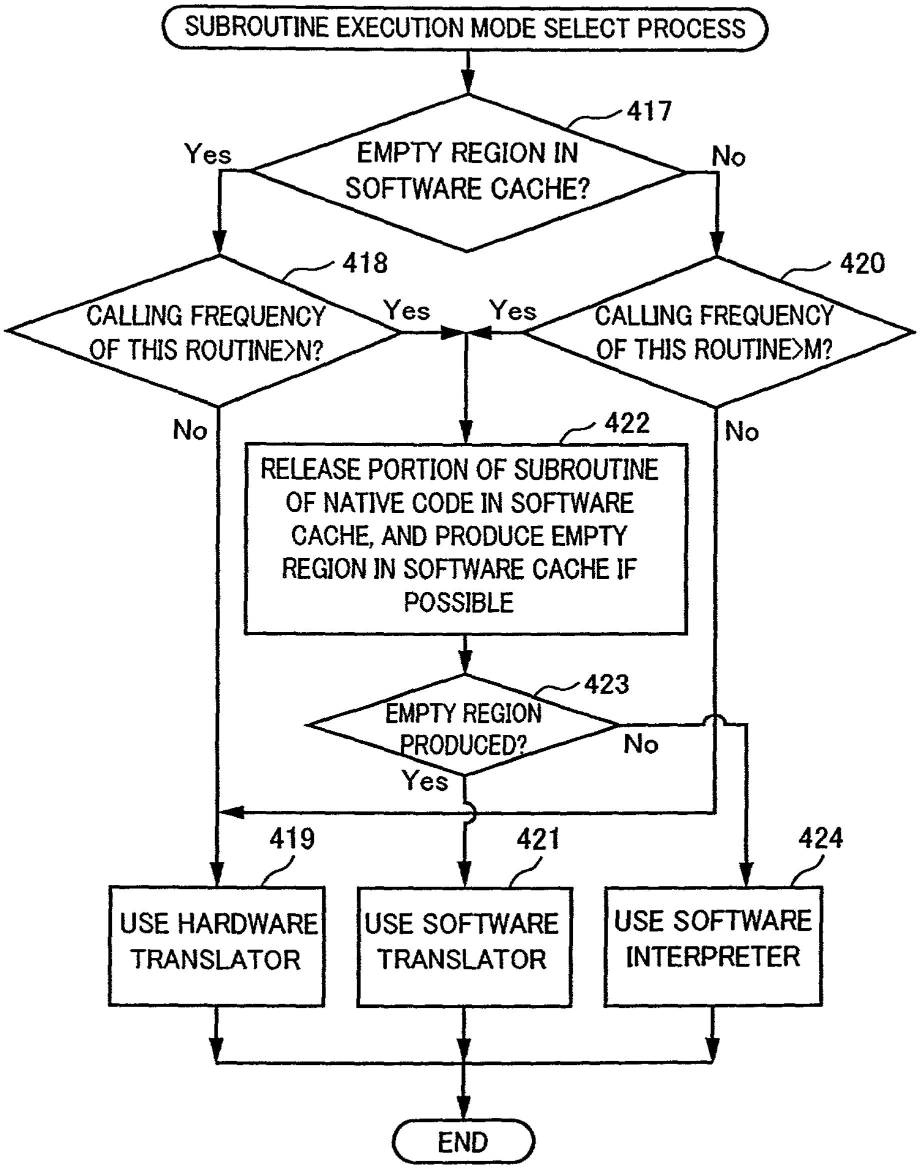

SUBROUTINE EXECUTION MODE SELECT PROCESS
417
EMPTY REGION IN SOFTWARE CACHE?
Yes
No
418
CALLING FREQUENCY OF THIS ROUTINE>N?
Yes
No
420
CALLING FREQUENCY OF THIS ROUTINE>M?
Yes
No
422
RELEASE PORTION OF SUBROUTINE OF NATIVE CODE IN SOFTWARE CACHE, AND PRODUCE EMPTY REGION IN SOFTWARE CACHE IF POSSIBLE
423
EMPTY REGION PRODUCED?
No
Yes
419
USE HARDWARE TRANSLATOR
421
USE SOFTWARE TRANSLATOR
424
USE SOFTWARE INTERPRETER
END

FIG. 10

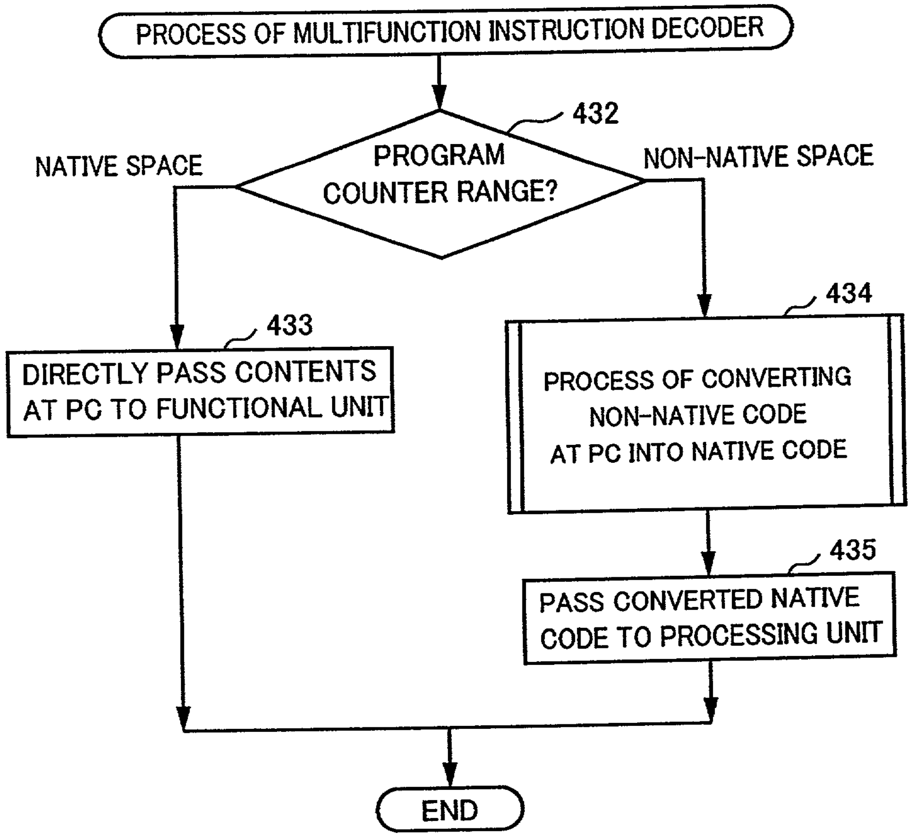

PROCESS OF MULTIFUNCTION INSTRUCTION DECODER
432
PROGRAM COUNTER RANGE?
NATIVE SPACE
NON-NATIVE SPACE
433
DIRECTLY PASS CONTENTS AT PC TO FUNCTIONAL UNIT
434
PROCESS OF CONVERTING NON-NATIVE CODE AT PC INTO NATIVE CODE
435
PASS CONVERTED NATIVE CODE TO PROCESSING UNIT
END

FIG. 11

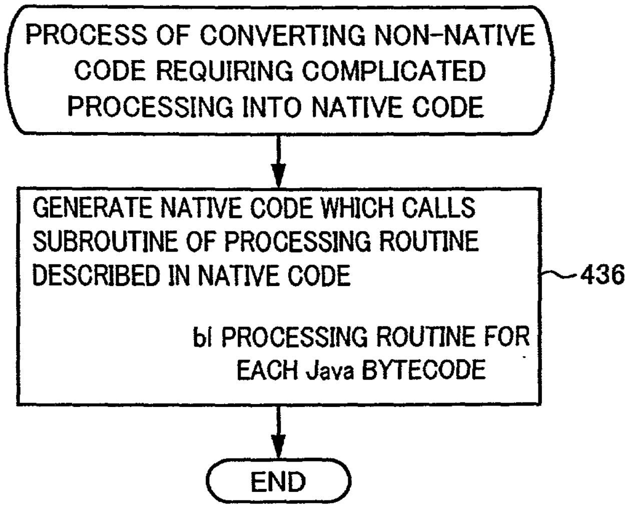

PROCESS OF CONVERTING NON-NATIVE CODE REQUIRING COMPLICATED PROCESSING INTO NATIVE CODE
GENERATE NATIVE CODE WHICH CALLS SUBROUTINE OF PROCESSING ROUTINE DESCRIBED IN NATIVE CODE
bl PROCESSING ROUTINE FOR EACH Java BYTECODE
436
END

FIG. 12

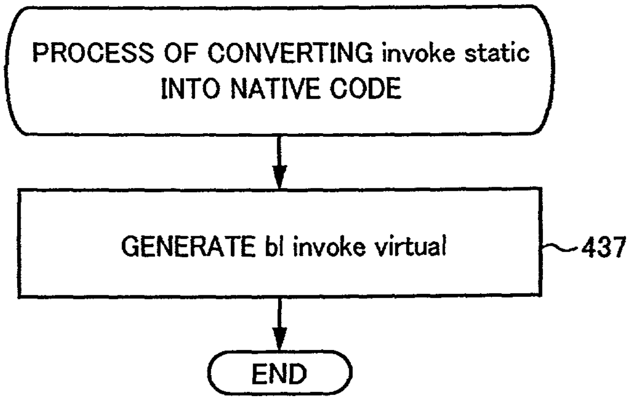

PROCESS OF CONVERTING invoke static INTO NATIVE CODE
GENERATE bl invoke virtual
437
END

FIG. 13

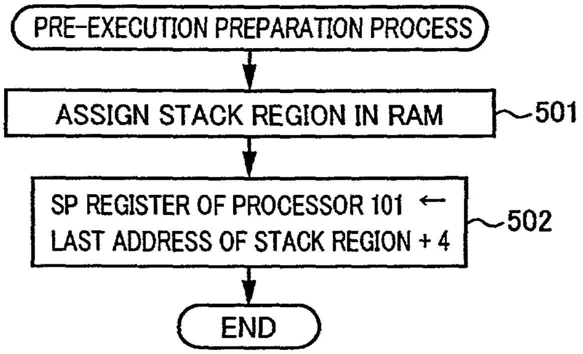

PRE-EXECUTION PREPARATION PROCESS
ASSIGN STACK REGION IN RAM
501
SP REGISTER OF PROCESSOR 101 ←
LAST ADDRESS OF STACK REGION + 4
502
END

FIG. 14

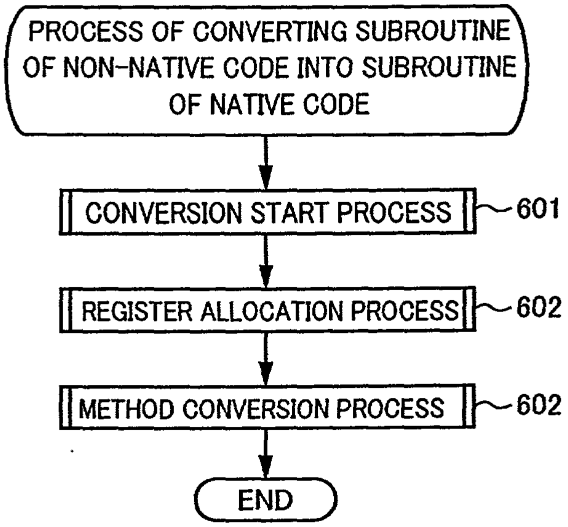

PROCESS OF CONVERTING SUBROUTINE OF NON-NATIVE CODE INTO SUBROUTINE OF NATIVE CODE
CONVERSION START PROCESS
601
REGISTER ALLOCATION PROCESS
602
METHOD CONVERSION PROCESS
602
END

FIG. 15

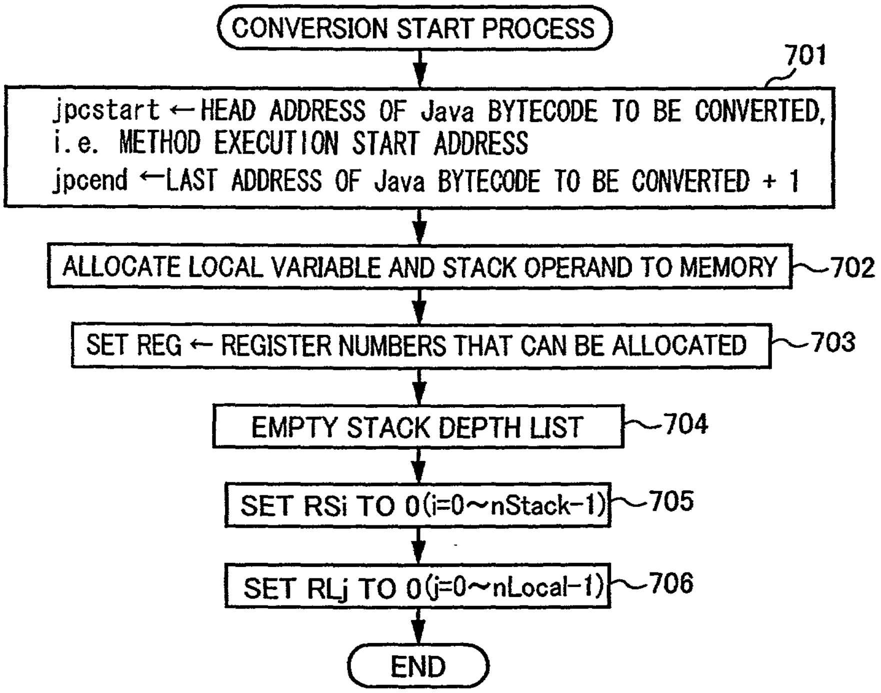

CONVERSION START PROCESS
701
jpcstart ←HEAD ADDRESS OF Java BYTECODE TO BE CONVERTED,
i.e. METHOD EXECUTION START ADDRESS
jpcend ←LAST ADDRESS OF Java BYTECODE TO BE CONVERTED + 1
ALLOCATE LOCAL VARIABLE AND STACK OPERAND TO MEMORY
702
SET REG ← REGISTER NUMBERS THAT CAN BE ALLOCATED
703
EMPTY STACK DEPTH LIST
704
SET RSi TO 0(i=0~nStack-1)
705
SET RLj TO 0(j=0~nLocal-1)
706
END

FIG. 16

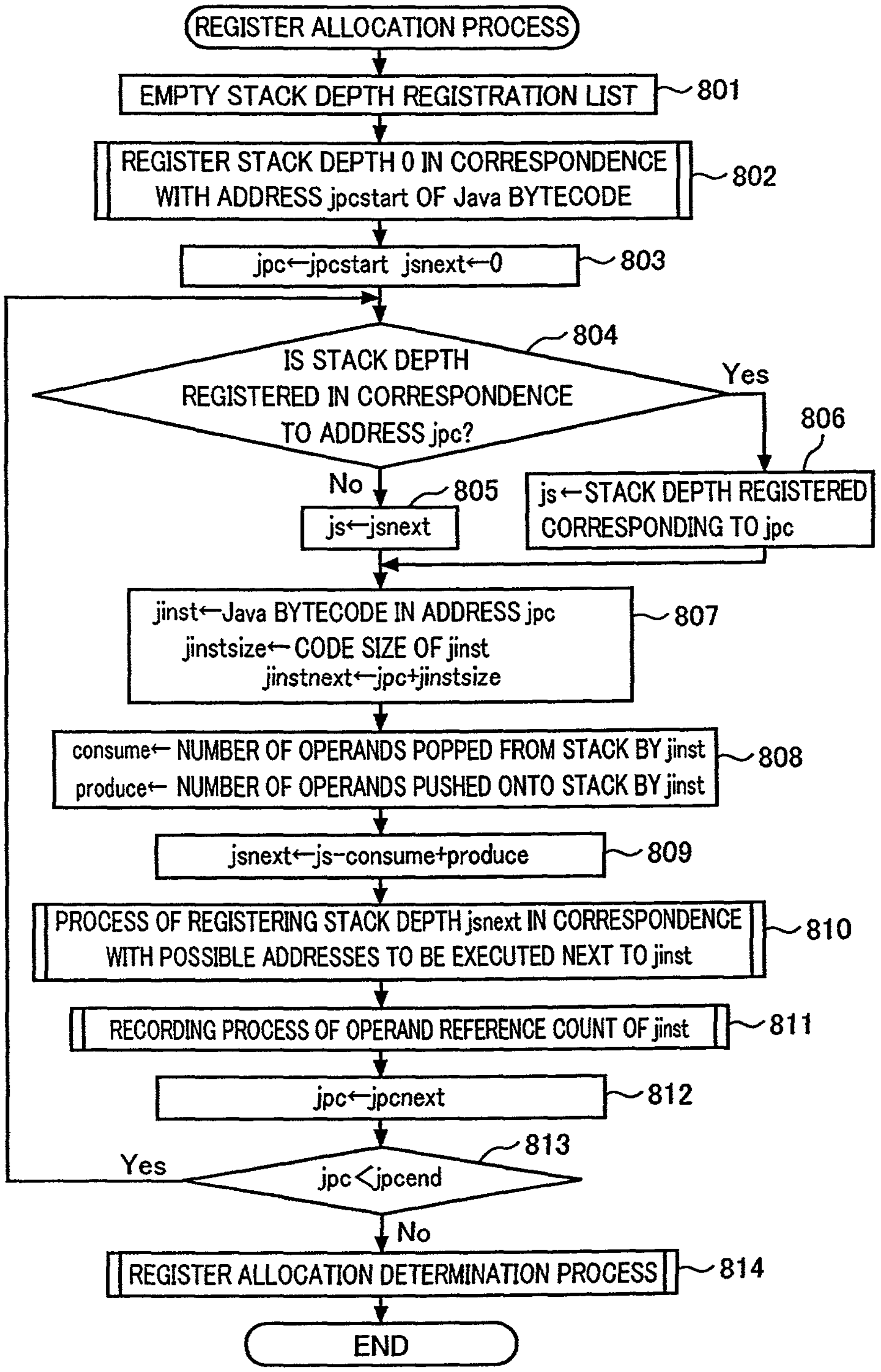

REGISTER ALLOCATION PROCESS
EMPTY STACK DEPTH REGISTRATION LIST
801
REGISTER STACK DEPTH 0 IN CORRESPONDENCE WITH ADDRESS jpcstart OF Java BYTECODE
802
jpc←jpcstart jsnext←0
803
IS STACK DEPTH REGISTERED IN CORRESPONDENCE TO ADDRESS jpc?
804
Yes
806
No
805
js←jsnext
js←STACK DEPTH REGISTERED CORRESPONDING TO jpc
jinst←Java BYTECODE IN ADDRESS jpc
jinstsize←CODE SIZE OF jinst
jinstnext←jpc+jinstsize
807
consume← NUMBER OF OPERANDS POPPED FROM STACK BY jinst
produce← NUMBER OF OPERANDS PUSHED ONTO STACK BY jinst
808
jsnext←js-consume+produce
809
PROCESS OF REGISTERING STACK DEPTH jsnext IN CORRESPONDENCE WITH POSSIBLE ADDRESSES TO BE EXECUTED NEXT TO jinst
810
RECORDING PROCESS OF OPERAND REFERENCE COUNT OF jinst
811
jpc←jpcnext
812
813
Yes
jpc<jpcend
No
REGISTER ALLOCATION DETERMINATION PROCESS
814
END

FIG. 17

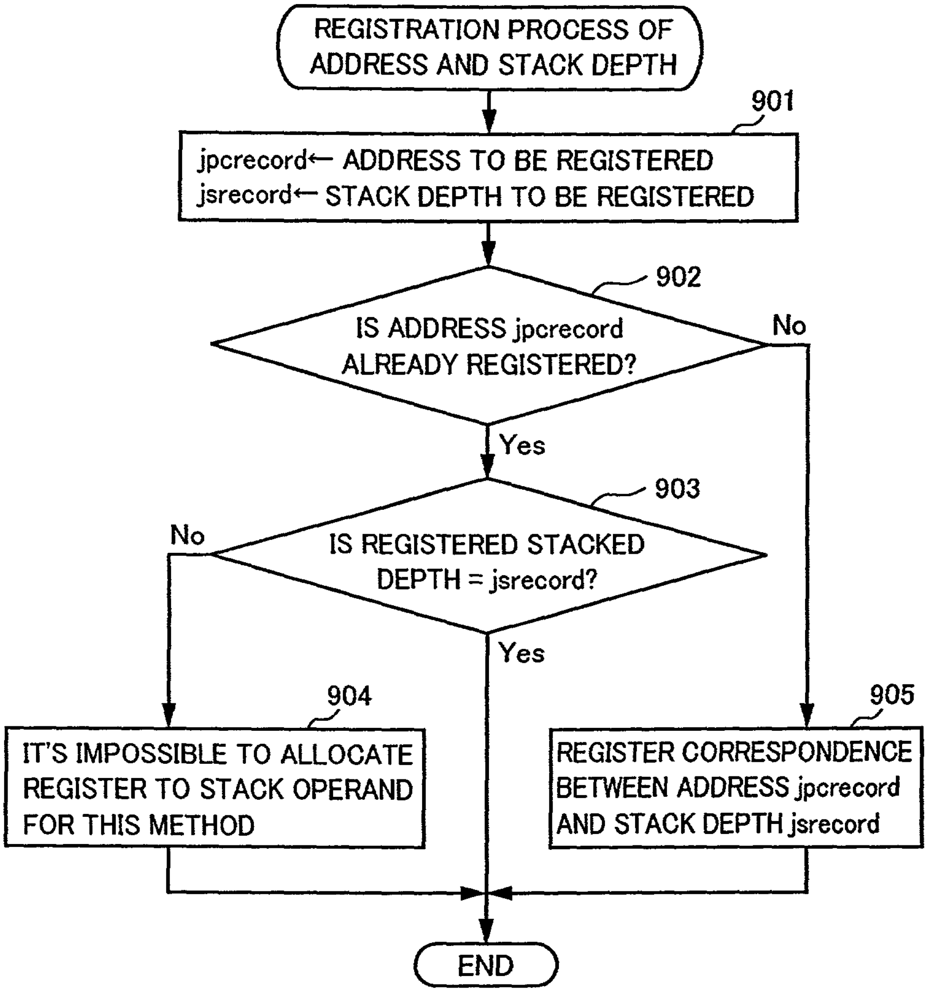

REGISTRATION PROCESS OF ADDRESS AND STACK DEPTH
901
jpcrecord← ADDRESS TO BE REGISTERED
jsrecord← STACK DEPTH TO BE REGISTERED
902
IS ADDRESS jpcrecord ALREADY REGISTERED?
No
Yes
903
IS REGISTERED STACKED DEPTH = jsrecord?
No
Yes
904
IT'S IMPOSSIBLE TO ALLOCATE REGISTER TO STACK OPERAND FOR THIS METHOD
905
REGISTER CORRESPONDENCE BETWEEN ADDRESS jpcrecord AND STACK DEPTH jsrecord
END

FIG. 18

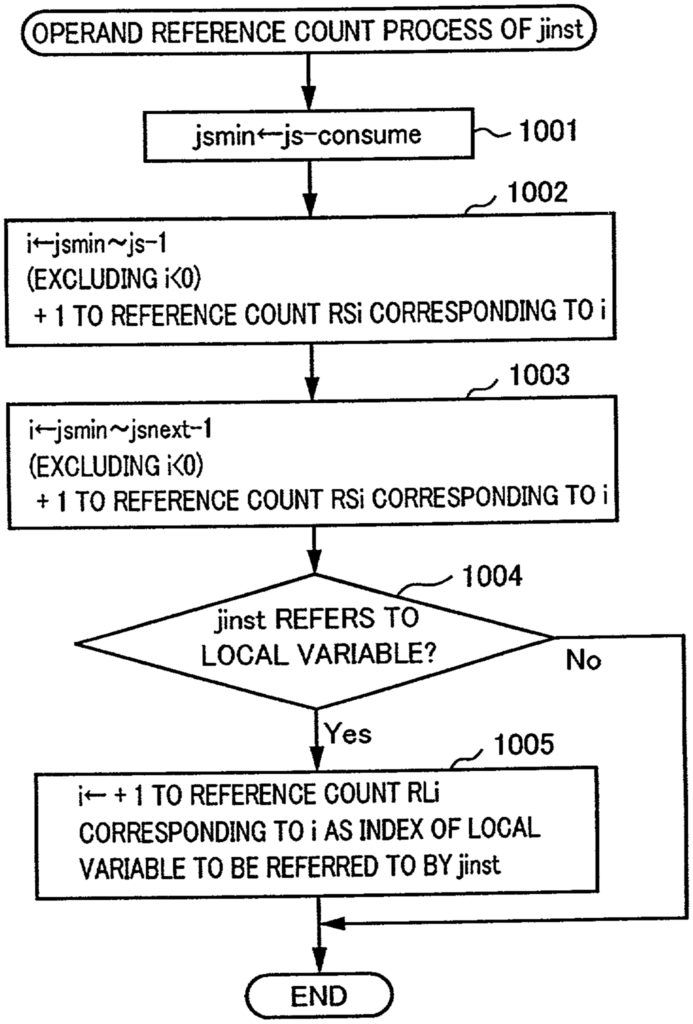

OPERAND REFERENCE COUNT PROCESS OF jinst
jsmin←js−consume
1001
1002
i←jsmin~js−1
(EXCLUDING i<0)
+ 1 TO REFERENCE COUNT RSi CORRESPONDING TO i
1003
i←jsmin~jsnext−1
(EXCLUDING i<0)
+ 1 TO REFERENCE COUNT RSi CORRESPONDING TO i
1004
jinst REFERS TO LOCAL VARIABLE?
No
Yes
1005
i← + 1 TO REFERENCE COUNT RLi
CORRESPONDING TO i AS INDEX OF LOCAL
VARIABLE TO BE REFERRED TO BY jinst
END

FIG. 19

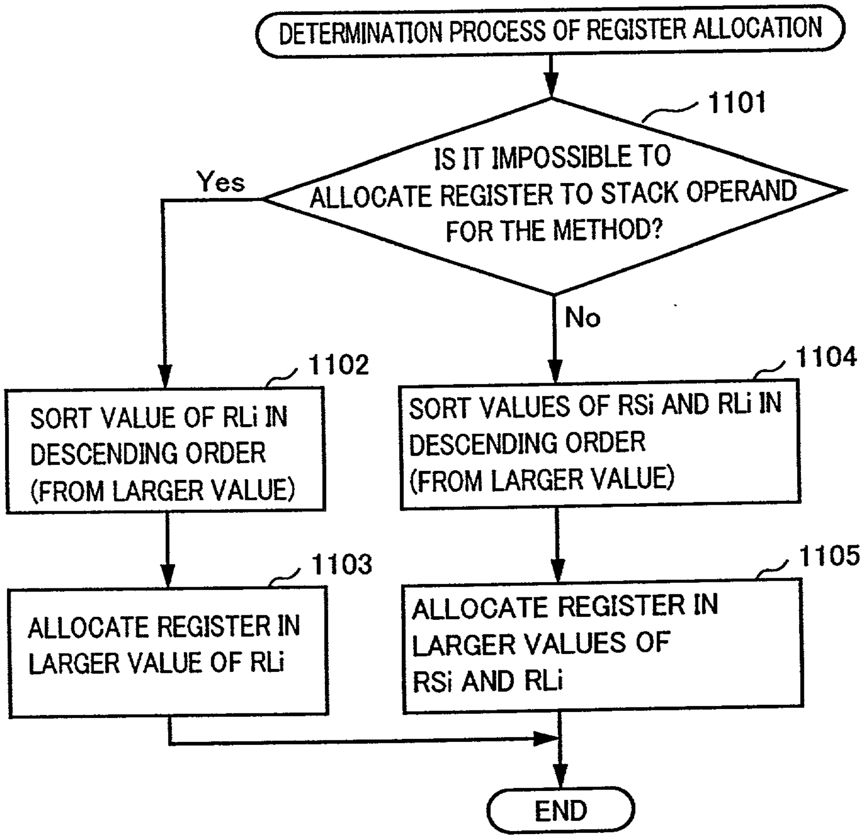

DETERMINATION PROCESS OF REGISTER ALLOCATION
1101
IS IT IMPOSSIBLE TO ALLOCATE REGISTER TO STACK OPERAND FOR THE METHOD?
Yes
No
1102
SORT VALUE OF RLi IN DESCENDING ORDER (FROM LARGER VALUE)
1103
ALLOCATE REGISTER IN LARGER VALUE OF RLi
1104
SORT VALUES OF RSi AND RLi IN DESCENDING ORDER (FROM LARGER VALUE)
1105
ALLOCATE REGISTER IN LARGER VALUES OF RSi AND RLi
END

FIG. 20

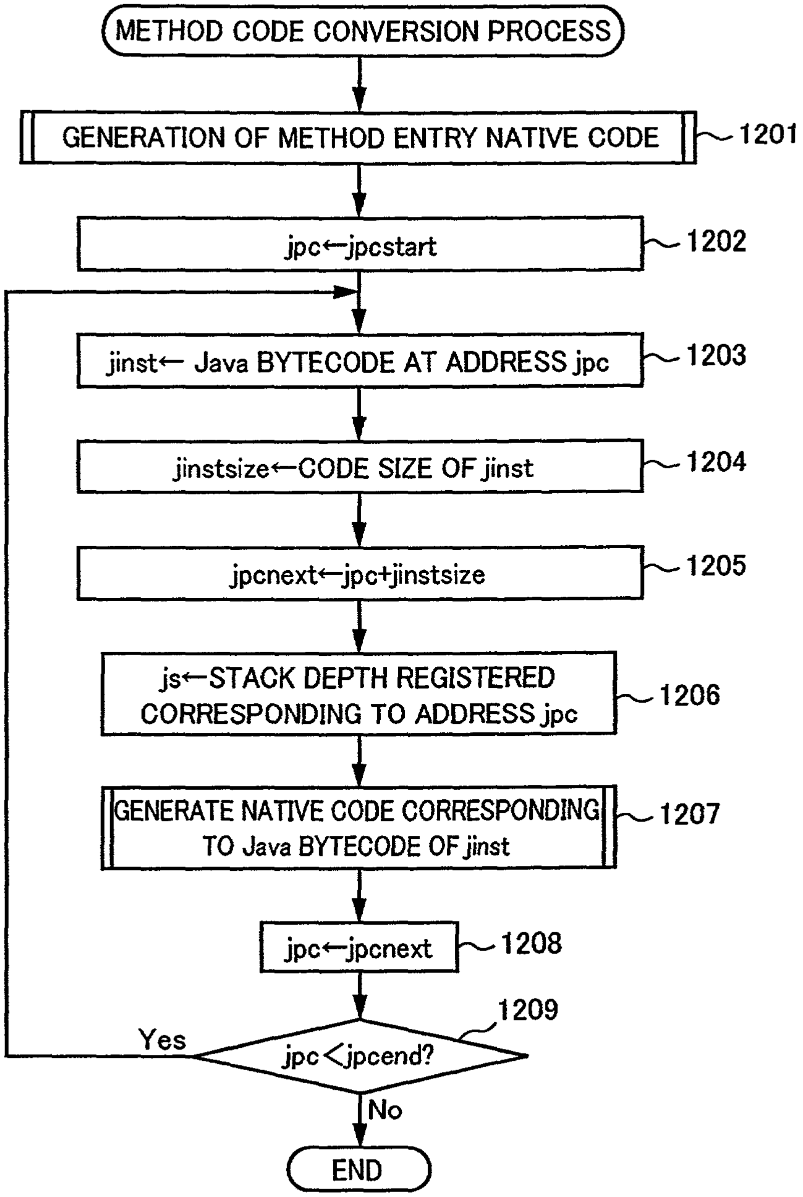

METHOD CODE CONVERSION PROCESS
GENERATION OF METHOD ENTRY NATIVE CODE
1201
jpc←jpcstart
1202
jinst← Java BYTECODE AT ADDRESS jpc
1203
jinstsize←CODE SIZE OF jinst
1204
jpcnext←jpc+jinstsize
1205
js←STACK DEPTH REGISTERED CORRESPONDING TO ADDRESS jpc
1206
GENERATE NATIVE CODE CORRESPONDING TO Java BYTECODE OF jinst
1207
jpc←jpcnext
1208
1209
Yes
jpc<jpcend?
No
END

FIG. 21

Java BYTECODE	OPERAND ALLOCATION	NATIVE CODE
	S<js>	
iconst_<n>	REGISTER	ldi S<js>, #n
	MEMORY	ldi r0, #n st r0, @S<js>

FIG. 22

Java BYTECODE	OPERAND ALLOCATION		NATIVE CODE
	S<js>	L<n>	
iload_<n>	REGISTER	REGISTER	mv S<js>, L<n>
	REGISTER	MEMORY	ld S<js>, @L<n>
	MEMORY	REGISTER	st L<n>, @S<js>
	MEMORY	MEMORY	ld r0, @L<n> st r0, @S<js>

FIG. 23

Java BYTECODE	OPERAND ALLOCATION		NATIVE CODE
	S<js-1>	L<n>	
istore_<n>	REGISTER	REGISTER	mv L<n>, S<js-1>
	REGISTER	MEMORY	st S<js-1>, @L<n>
	MEMORY	REGISTER	ld L<n>, @S<js-1>
	MEMORY	MEMORY	ld r0, @S<js-1> st r0, @L<n>

FIG. 24

Java BYTECODE	OPERAND ALLOCATION		OPERAND ALLOCATION
	S<js-2>	S<js-1>	
iadd	REGISTER	REGISTER	add S<js-2>, S<js-1>
	REGISTER	MEMORY	ld r0, @S<js-1> add S<js-2>, r0
	MEMORY	REGISTER	ld r0, @S<js-2> add r0, S<js-1> st r0, @S<js-2>
	MEMORY	MEMORY	ld r0, @S<js-2> ld r1, @S<js-1> add r0, r1 st r0, @S<js-2>

FIG. 25

Java BYTECODE	OPERAND ALLOCATION	NATIVE CODE
	S<js-1>	
ifge X	REGISTER	bgez S<js-1>, TX
	MEMORY	ld r0, @S<js-1> bgez r0, TX

TX IS ADDRESS OF NATIVE CODE GENERATED FOR Java BYTECODE OF ADDRESS X

FIG. 26

Java BYTECODE	–	NATIVE CODE
goto X	–	bra TX

TX IS ADDRESS OF NATIVE CODE GENERATED FOR Java BYTECODE OF ADDRESS X

FIG. 27

Java BYTECODE	OPERAND ALLOCATION	NATIVE CODE
	S<js-1>	
ireturn	REGISTER	mv r0, S<js-1>
	MEMORY	ld r0, @S<js-1>

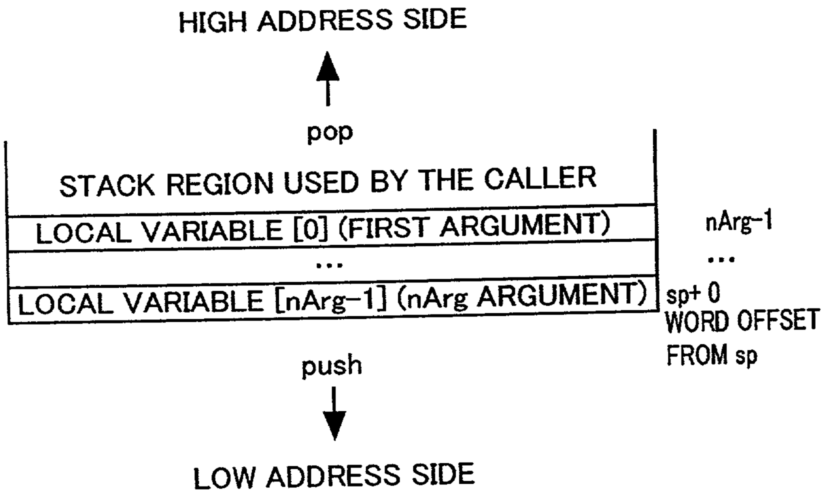
FIG. 28

Java BYTECODE	OPERAND ALLOCATION		NATIVE CODE
	S<js-2>	S<js-1>	
intokestatic <int F(int, int)>	REGISTER	REGISTER	push S<js-2> push S<js-1> ld24 r0, #method_id bl call_java_method addi sp, #8 mv S<js-1>, r0
	REGISTER	MEMORY	push S<js-2> ld r0, @S<js-1> push r0 ld24 r0, #method_id bl call_java_method addi sp, #8 mv S<js-1>, r0
	MEMORY	REGISTER	ld r0, @S<js-2> push r0 push S<js-1> ld24 r0, #method_id bl call_java_method addi sp, #8 st r0, @S<js-1>
	MEMORY	MEMORY	ld r0, @S<js-2> push r0 ld r0, @S<js-1> push r0 ld24 r0, #method_id bl call_java_method addi sp, #8 st r0, @S<js-1>

FIG. 29

REGISTER	USAGE
r0-r3	MAY BE USED TEMPORARILY FOR CALCULATION r0 AND r1 ARE USED TO STORE RETURN VALUE IN RETURNING FROM SUBROUTINE VALUES OF THESE REGISTERS ARE NOT PRESERVED ACROSS SUBROUTINE CALL
r4-r7	MAY BE USED TEMPORARILY FOR CALCULATION VALUES OF THESE REGISTERS ARE NOT PRESERVED ACROSS SUBROUTINE CALL
r8-r13	ALLOCATED FOR OPERAND STACK AND LOCAL VARIABLE VALUES OF THESE REGISTERS ARE PRESERVED ACROSS SUBROUTINE CALL
r14(lr)	LINK REGISTER USED TO STORE RETURN ADDRESS IN SUBROUTINE CALL VALUE OF THIS REGISTER IS NOT PRESERVED ACROSS SUBROUTINE CALL
r15(sp)	STACK POINTER VALUE OF THIS REGISTER IS PRESERVED ACROSS SUBROUTINE CALL

FIG. 30


HIGH ADDRESS SIDE
pop
STACK REGION USED BY THE CALLER
LOCAL VARIABLE [0] (FIRST ARGUMENT)
nArg-1
...
...
LOCAL VARIABLE [nArg-1] (nArg ARGUMENT)
sp+ 0
WORD OFFSET
FROM sp
push
LOW ADDRESS SIDE

NATIVE STACK BEFORE EXECUTING
METHOD ENTRY CODE

FIG. 31

HIGH ADDRESS SIDE

↑

pop

STACK REGION USED BY THE CALLER	
LOCAL VARIABLE [0] (FIRST ARGUMENT)	7+nStack+nLocal-1
...	...
LOCAL VARIABLE [nArg-1] (nArg ARGUMENT)	7+nStack+nLocal-nArg
LOCAL VARIABLE [nArg]	7+nStack+nLocal-nArg-1
...	...
LOCAL VARIABLE [nLocal-1]	7+nStack
STACK OPERAND [0]	7+nStack-1
...	...
STACK OPERAND [nStack-1]	7
OLD r8	6
OLD r9	5
OLD r10	4
OLD r11	3
OLD r12	2
OLD r13	1
OLD r14 (RETURN ADDRESS)	sp+ 0
	WORD OFFSET FROM sp

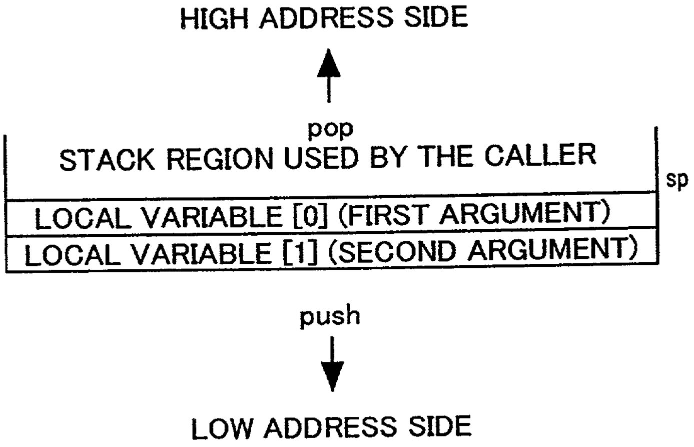
push

↓

LOW ADDRESS SIDE

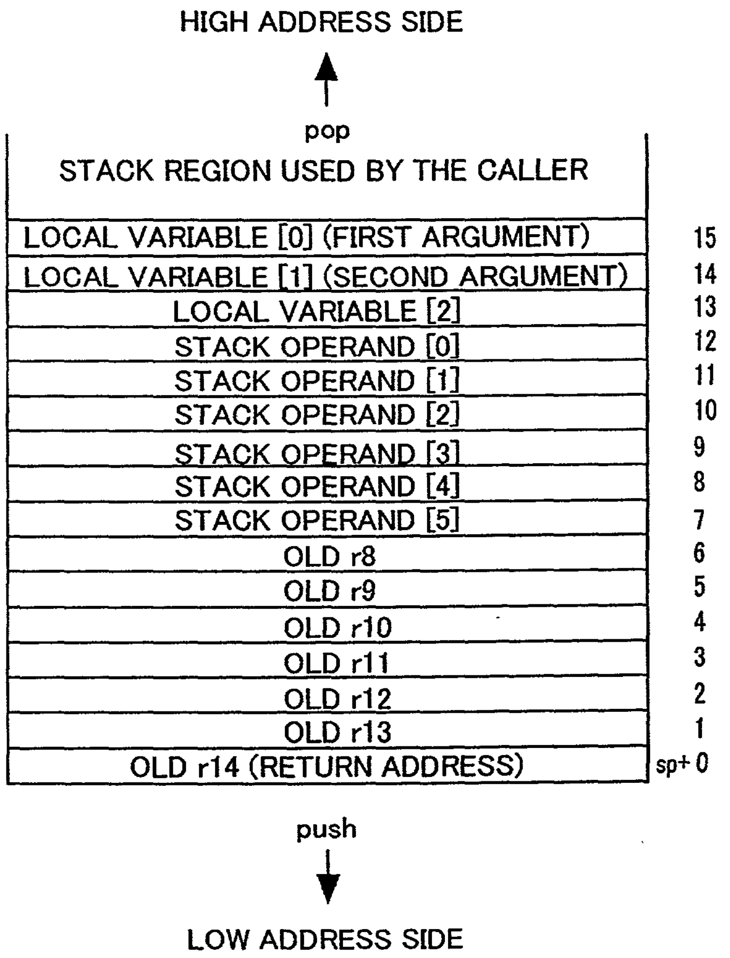
NATIVE STACK AFTER EXECUTING
METHOD ENTRY CODE

FIG. 32


HIGH ADDRESS SIDE
pop
STACK REGION USED BY THE CALLER
sp
LOCAL VARIABLE [0] (FIRST ARGUMENT)
LOCAL VARIABLE [1] (SECOND ARGUMENT)
push
LOW ADDRESS SIDE

NATIVE STACK BEFORE EXECUTING
METHOD ENTRY CODE
nStack=6, nLocal=2, nArg=2

FIG. 33



NATIVE STACK AFTER EXECUTING
METHOD ENTRY CODE
nStack=6, nLocal=3, nArg=2

FIG. 34

SYMBOL	OPERAND	REGISTER ALLOCATION	
		IMMEDIATELY AFTER CONVERSION START PROCESS	AFTER REGISTER ALLOCATION PROCESS
L<0>	LOCAL VARIABLE [0] (FIRST ARGUMENT)	(32 , SP)	R 13
L<1>	LOCAL VARIABLE [1] (SECOND ARGUMENT)	(28 , SP)	(56 , SP)
L<2>	LOCAL VARIABLE [2]	(24 , SP)	(52 , SP)
S<0>	STACK OPERAND[0]	(20 , SP)	R 9
S<1>	STACK OPERAND[1]	(16 , SP)	R 8
S<2>	STACK OPERAND[2]	(12 , SP)	R 10
S<3>	STACK OPERAND[3]	(8 , SP)	R 11
S<4>	STACK OPERAND[4]	(4 , SP)	R 12
S<5>	STACK OPERAND[5]	(0 , SP)	(28 , SP)

FIG. 35

STATUS	jpc	js	jinst	jinstsize	jpcnext	consume	produce	jsnext	INSTRUCTION ADDRESS AND STACK DEPTH REGISTERED IN STACK DEPTH LIST	RS						RL		
										0	1	2	3	4	5	0	1	2
(1)	0							0	[0,0]	0	0	0	0	0	0	0	0	0
(2)	0	0	iload_0	1	1	0	1	1	[1,1]	1	0	0	0	0	0	1	0	0
(3)	1	1	iload_1	1	2	0	1	2	[2,2]	1	1	0	0	0	0	1	1	0
(4)	2	2	iadd	1	3	2	1	1	[3,1]	3	2	0	0	0	0	1	1	0
(5)	3	1	istore_2	1	4	1	0	0	[4,0]	4	2	0	0	0	0	1	1	1
(6)	4	0	iconst_1	1	5	0	1	1	[5,1]	5	2	0	0	0	0	1	1	1
(7)	5	1	iload_0	1	6	0	1	2	[6,2]	5	3	0	0	0	0	2	1	1
(8)	6	2	ifge 21	3	9	1	0	1	[9,1][21,1]	5	4	0	0	0	0	2	1	1
(9)	9	1	iconst_2	1	10	0	1	2	[10,2]	5	5	0	0	0	0	2	1	1
(10)	10	2	iload_0	1	11	0	1	3	[11,3]	5	5	1	0	0	0	3	1	1
(11)	11	3	iload_1	1	12	0	1	4	[12,4]	5	5	1	1	0	0	3	2	1
(12)	12	4	iconst_3	1	13	0	1	5	[13,5]	5	5	1	1	1	0	3	2	1
(13)	13	5	iload_2	1	14	0	1	6	[14,6]	5	5	1	1	1	1	3	2	2
(14)	14	6	iadd	1	15	2	1	5	[15,5]	5	5	1	1	3	2	3	2	2
(15)	15	5	idiv	1	16	2	1	4	[16,4]	5	5	1	3	4	2	3	2	2
(16)	16	4	iadd	1	17	2	1	3	[17,3]	5	5	3	4	4	2	3	2	2
(17)	17	3	imul	1	18	2	1	2	[18,2]	5	7	4	4	4	2	3	2	2
(18)	18	2	goto 28	3	21	0	0	2	[28,2]	5	7	4	4	4	2	3	2	2

FIG. 36

STATUS	jpc	js	jinst	jinstsize	jpcnext	consume	produce	jsnext	INSTRUCTION ADDRESS AND STACK DEPTH REGISTERED IN STACK DEPTH LIST	RS						RL		
										0	1	2	3	4	5	0	1	2
(19)	21	1	iload_0	1	22	0	1	2	[22,2]	5	8	4	4	4	2	4	2	2
(20)	22	2	iconst_1	1	23	0	1	3	[23,3]	5	8	5	4	4	2	4	2	2
(21)	23	3	isub	1	24	2	1	2	[24,2]	5	10	6	4	4	2	4	2	2
(22)	24	2	iload_2	1	25	0	1	3	[25,3]	5	10	7	4	4	2	4	2	3
(23)	25	3	invokestati c <int f(int,	3	28	2	1	2	[28,2]	5	12	8	4	4	2	4	2	3
(24)	28	2	iadd	1	29	2	1	1	[29,1]	7	13	8	4	4	2	4	2	3
(25)	29	1	ireturn	1	30	1	0	0		8	13	8	4	4	2	4	2	3

FIG. 37

STATUS	jpc	jinst	jinstsize	jpcnext	js	NATIVE CODE
(1)						addi sp, #-(nLocal-nArg+nStack)*4 push r8 push r9 push r10 push r11 push r12 push r13 push lr ld L<0>, @((nLocal+nStack+nSave-1)*4,sp)
(2)	0	iload_0	1	1	0	mv S<0>, L<0>
(3)	1	iload_1	1	2	1	ld S<1>, @L<1>
(4)	2	iadd	1	3	2	add S<0>, S<1>
(5)	3	istore_2	1	4	1	st S<0>, @L<2>
(6)	4	iconst_1	1	5	0	ldi S<0>, #1
(7)	5	iload_0	1	6	1	mv S<1>, L<0>
(8)	6	ifge 21	3	9	2	bgez S<1>, T21
(9)	9	iconst_2	1	10	1	ldi S<1>, #2
(10)	10	iload_0	1	11	2	mv S<2>, L<0>
(11)	11	iload_1	1	12	3	ld S<3>, @L<1>
(12)	12	iconst_3	1	13	4	ldi S<4>, #3
(13)	13	iload_2	1	14	5	ld r0, @L<2> st r0, @S<5>
(14)	14	iadd	1	15	6	ld r0, @S<5> add S<4>, r0
(15)	15	idiv	1	16	5	div S<3>, S<4>
(16)	16	iadd	1	17	4	add S<2>, S<3>
(17)	17	imul	1	18	3	mul S<1>, S<2>
(18)	18	goto 28	3	21	2	bra T28
(19)	21	iload_0	1	22	1	T21: mv S<1>, L<0>
(20)	22	iconst_1	1	23	2	ldi S<2>, #1
(21)	23	isub	1	24	3	sub S<1>, S<2>

FIG. 38

STATUS	jpc	jinst	jinstsize	jpcnext	js	NATIVE CODE
(22)	24	iload_2	1	25	2	ld S<2>, @L<2>
(23)	25	invokestatic <int F(int, int)>	3	28	3	push S<2> push S<1> ld24 r0, #methodId jl callJavaMethod addi sp, #8 mv S<1>, r0
(24)	28	iadd	1	29	2	T28: add S<0>, S<1>
(25)	29	ireturn	1	30	1	mv r0, S<0> pop lr pop r13 pop r12 pop r11 pop r10 pop r9 pop r8 addi sp, #(nLocal-nArg+nStack)*4 jmp lr

FIG. 39

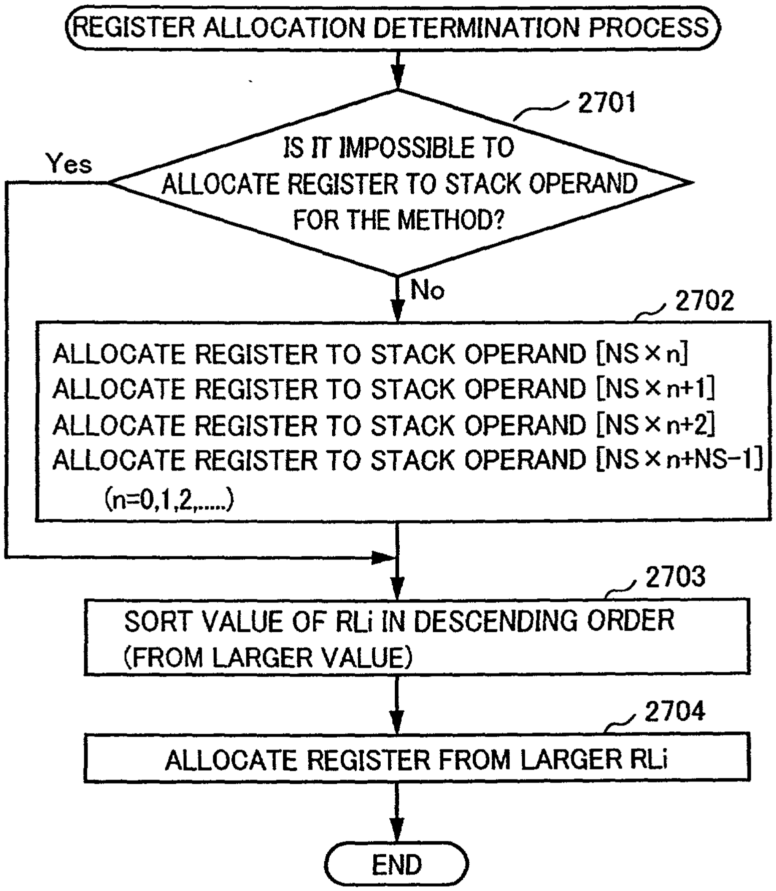

REGISTER ALLOCATION DETERMINATION PROCESS
2701
IS IT IMPOSSIBLE TO ALLOCATE REGISTER TO STACK OPERAND FOR THE METHOD?
Yes
No
2702
ALLOCATE REGISTER TO STACK OPERAND [NS × n]
ALLOCATE REGISTER TO STACK OPERAND [NS × n+1]
ALLOCATE REGISTER TO STACK OPERAND [NS × n+2]
ALLOCATE REGISTER TO STACK OPERAND [NS × n+NS−1]
(n=0,1,2,.....)
2703
SORT VALUE OF RLi IN DESCENDING ORDER (FROM LARGER VALUE)
2704
ALLOCATE REGISTER FROM LARGER RLi
END

FIG. 40

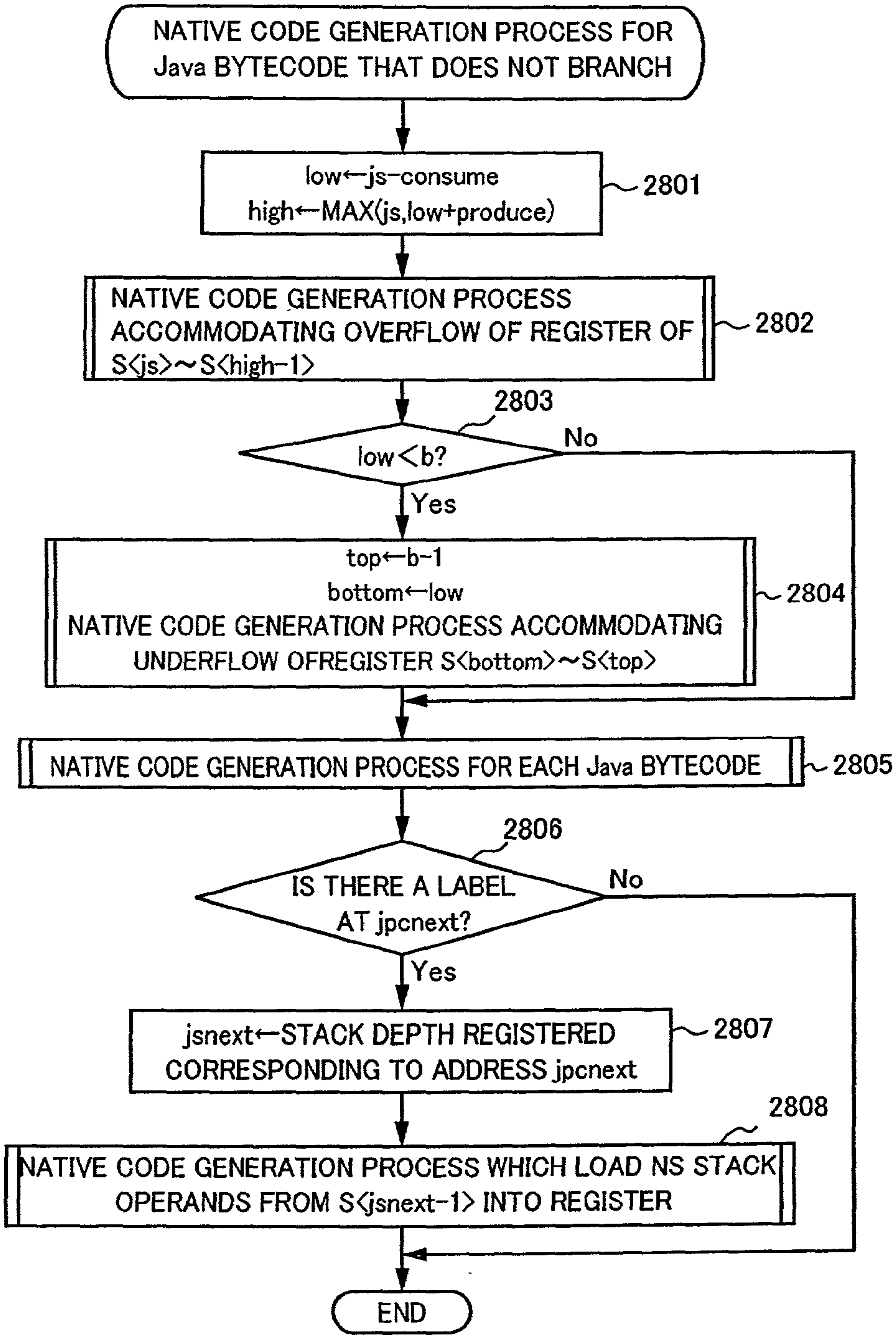

NATIVE CODE GENERATION PROCESS FOR Java BYTECODE THAT DOES NOT BRANCH
low←js−consume
high←MAX(js,low+produce)
2801
NATIVE CODE GENERATION PROCESS ACCOMMODATING OVERFLOW OF REGISTER OF S<js>~S<high−1>
2802
2803
low<b?
No
Yes
top←b−1
bottom←low
NATIVE CODE GENERATION PROCESS ACCOMMODATING UNDERFLOW OFREGISTER S<bottom>~S<top>
2804
NATIVE CODE GENERATION PROCESS FOR EACH Java BYTECODE
2805
2806
IS THERE A LABEL AT jpcnext?
No
Yes
jsnext←STACK DEPTH REGISTERED CORRESPONDING TO ADDRESS jpcnext
2807
2808
NATIVE CODE GENERATION PROCESS WHICH LOAD NS STACK OPERANDS FROM S<jsnext−1> INTO REGISTER
END

FIG. 41

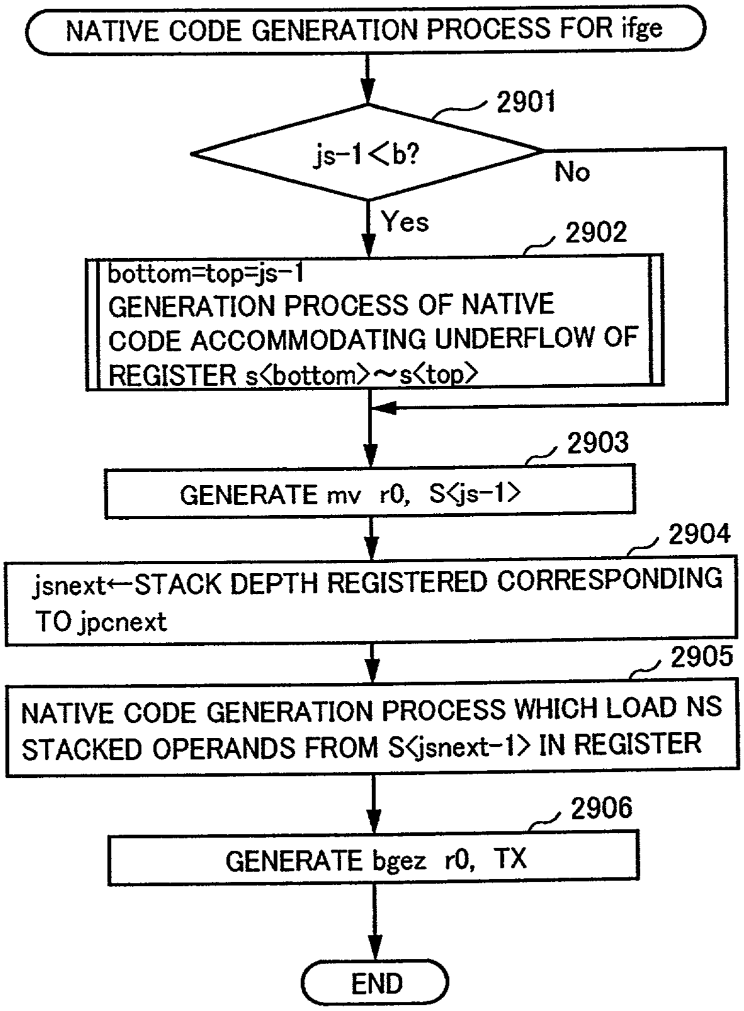

NATIVE CODE GENERATION PROCESS FOR ifge
2901
js-1<b?
No
Yes
2902
bottom=top=js-1
GENERATION PROCESS OF NATIVE CODE ACCOMMODATING UNDERFLOW OF REGISTER s<bottom>~s<top>
2903
GENERATE mv r0, S<js-1>
2904
jsnext←STACK DEPTH REGISTERED CORRESPONDING TO jpcnext
2905
NATIVE CODE GENERATION PROCESS WHICH LOAD NS STACKED OPERANDS FROM S<jsnext-1> IN REGISTER
2906
GENERATE bgez r0, TX
END

FIG. 42

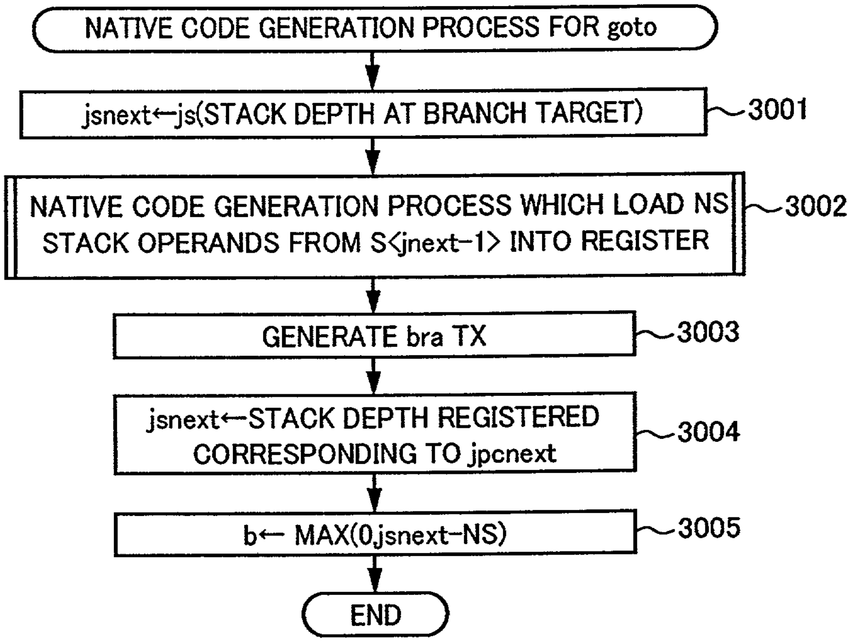

NATIVE CODE GENERATION PROCESS FOR goto
jsnext←js(STACK DEPTH AT BRANCH TARGET)
3001
NATIVE CODE GENERATION PROCESS WHICH LOAD NS STACK OPERANDS FROM S<jnext-1> INTO REGISTER
3002
GENERATE bra TX
3003
jsnext←STACK DEPTH REGISTERED CORRESPONDING TO jpcnext
3004
b← MAX(0,jsnext-NS)
3005
END

FIG. 43

REGISTER	USAGE
r0–r3	MAY BE USED TEMPORARILY FOR CALCULATION r0 AND r1 ARE USED TO STORE RETURN VALUE IN RETURNING FROM SUBROUTINE VALUES OF THESE REGISTERS ARE NOT PRESERVED ACROSS SUBORDINATE CALL
r4–r7	MAY BE USED TEMPORARILY FOR CALCULATION VALUES OF THESE REGISTERS ARE NOT PRESERVED ACROSS SUBORDINATE CALL
r8–r13	ALLOCATED FOR OPERAND STACK ALLOCATE OPERAND STACK [4n] TO r8 ALLOCATE OPERAND STACK [4n+1] TO r9 ALLOCATE OPERAND STACK [4n+2] TO r10 ALLOCATE OPERAND STACK [4n+3] TO r11 (n=0,1,2...) VALUES OF THESE REGISTERS ARE PRESERVED ACROSS SUBORDINATE CALL
r12–r13	ALLOCATED FOR LOCAL VARIABLES VALUES OF THESE REGISTERS ARE PRESERVED ACROSS SUBORDINATE CALL
r14(lr)	LINK REGISTER USED TO STORE RETURN ADDRESS IN SUBROUTINE CALL VALUE OF THIS REGISTER IS NOT PRESERVED ACROSS SUBORDINATE CALL
r15(sp)	STACK POINTER VALUE OF THIS REGISTER IS PRESERVED ACROSS SUBORDINATE CALL

FIG. 44

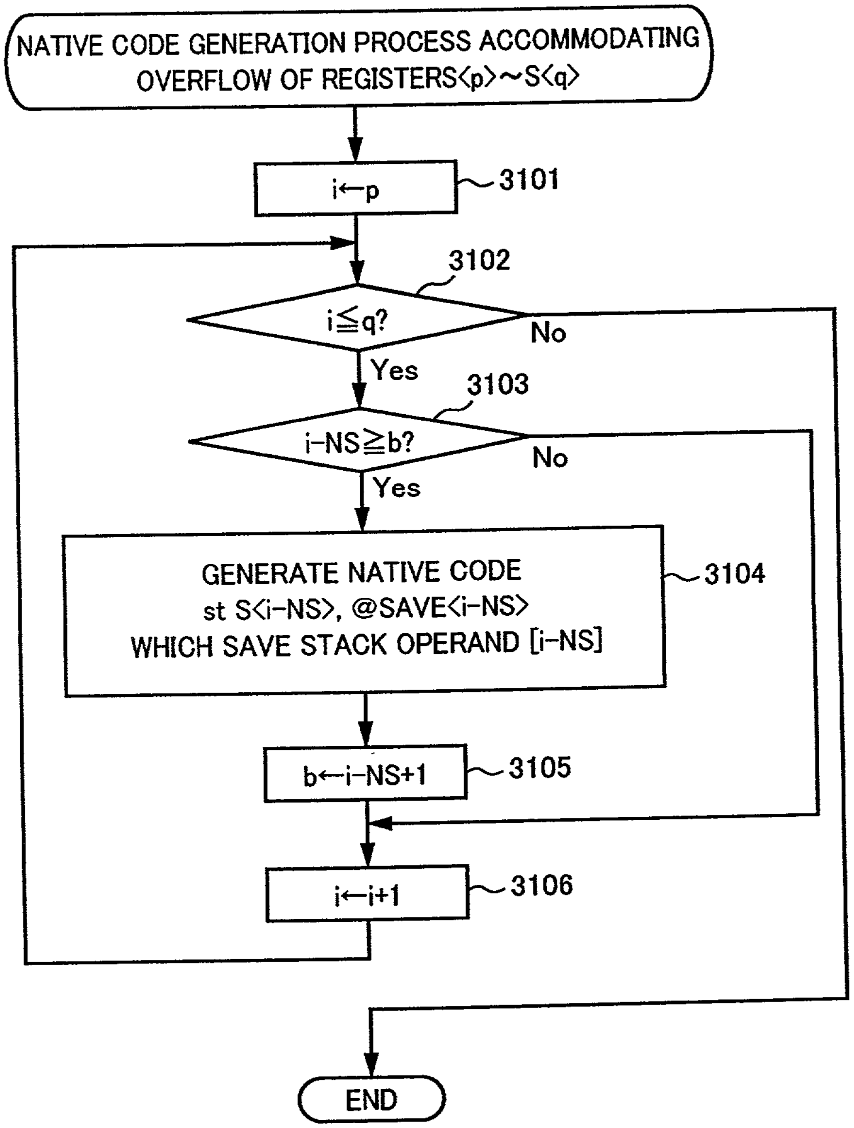

NATIVE CODE GENERATION PROCESS ACCOMMODATING OVERFLOW OF REGISTERS<p>~S<q>
i←p
3101
3102
i≦q?
No
Yes
3103
i-NS≧b?
No
Yes
GENERATE NATIVE CODE
st S<i-NS>, @SAVE<i-NS>
WHICH SAVE STACK OPERAND [i-NS]
3104
b←i-NS+1
3105
i←i+1
3106
END

FIG. 45

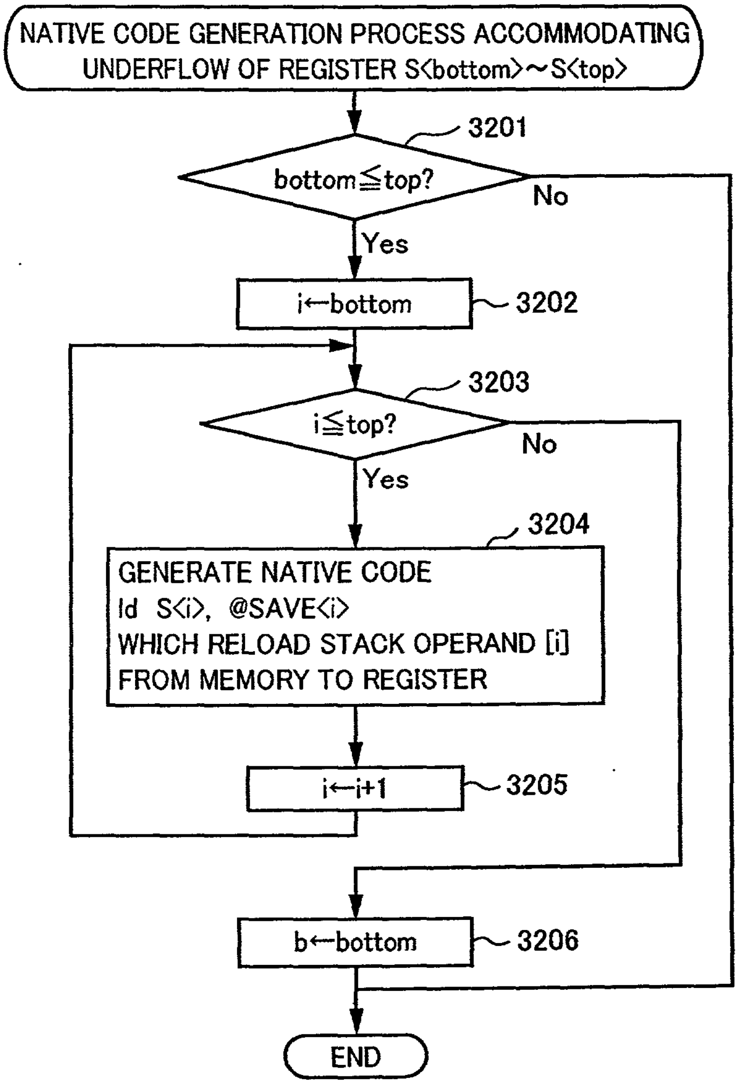

NATIVE CODE GENERATION PROCESS ACCOMMODATING UNDERFLOW OF REGISTER S<bottom>~S<top>
3201
bottom≦top?
No
Yes
i←bottom
3202
3203
i≦top?
No
Yes
3204
GENERATE NATIVE CODE
ld S<i>, @SAVE<i>
WHICH RELOAD STACK OPERAND [i]
FROM MEMORY TO REGISTER
i←i+1
3205
b←bottom
3206
END

FIG. 46

SYMBOL	OPERAND	REGISTER ALLOCATION	
		IMMEDIATELY AFTER CONVERSION START PROCESS	AFTER REGISTER ALLOCATION PROCESS
L<0>	LOCAL VARIABLE [0] (FIRST ARGUMENT)	(60 ,SP)	R12
L<1>	LOCAL VARIABLE [1] (SECOND ARGUMENT)	(56 ,SP)	(56 ,SP)
L<2>	LOCAL VARIABLE [2]	(52 ,SP)	R13
S<0>	STACK OPERAND[0]	(48 ,SP)	R8
S<1>	STACK OPERAND[1]	(44 ,SP)	R9
S<2>	STACK OPERAND[2]	(40 ,SP)	R10
S<3>	STACK OPERAND[3]	(36 ,SP)	R11
S<4>	STACK OPERAND[4]	(32 ,SP)	R8
S<5>	STACK OPERAND[5]	(28 ,SP)	R9

FIG. 47

STATUS	jpc	jinst	jinstsize	jpcnext	js	b	consume	produce	low	high	NATIVE CODE
(1)	0					0					addi sp, #-(nLocal-nArg+nStack)*4 push r8 push r9 push r10 push r11 push r12 push r13 push lr ld L<0>, @((nLocal+nStack+nSave-1)*4,sp) ld L<1>, @((nLocal+nStack+nSave-2)*4,sp)
(2)	0	iload_0	1	1	0	0	0	1	0	1	mv S<0>, L<0>
(3)	1	iload_1	1	2	1	0	0	1	1	2	mv S<1>, L<1>
(4)	2	iadd	1	3	2	0	2	1	0	2	add S<0>, S<1>
(5)	3	istore_2	1	4	1	0	1	0	0	1	st S<0>, @L<2>
(6)	4	iconst_1	1	5	0	0	0	1	0	1	ldi S<0>, #1
(7)	5	iload_0	1	6	1	0	0	1	1	2	mv S<1>, L<0>
(8)	6	ifge 21	3	9	2	0	1	0	1	2	bgez S<1>, T21
(9)	9	iconst_2	1	10	1	0	0	1	1	2	ldi S<1>, #2
(10)	10	iload_0	1	11	2	0	0	1	2	3	mv S<2>, L<0>
(11)	11	iload_1	1	12	3	0	0	1	3	4	ld S<3>, @L<1>
(12)	12	iconst_3	1	13	4	1	0	1	4	5	st S<0>, @SAVE<0> ldi S<4>, #3
(13)	13	iload_2	1	14	5	2	0	1	5	6	st S<1>, @SAVE<1> ld S<5>, @L<2>
(14)	14	iadd	1	15	6	2	2	1	4	6	add S<4>, S<5>
(15)	15	idiv	1	16	5	2	2	1	3	5	div S<3>, S<4>
(16)	16	iadd	1	17	4	2	2	1	2	4	add S<2>, S<3>
(17)	17	imul	1	18	3	1	2	1	1	3	ld S<1>, @SAVE<1> mul S<1>, S<2>
(18)	18	goto 28	3	21	2	0	0	0	2	2	ld S<0>, @SAVE<0> bra T28
(19)	21	iload_0	1	22	1	0	0	1	1	2	T21: mv S<1>, L<0>
(20)	22	iconst_1	1	23	2	0	0	1	2	3	ldi S<2>, #1
(21)	23	isub	1	24	3	0	2	1	1	3	sub S<1>, S<2>
(22)	24	iload_2	1	25	2	0	0	1	2	3	ld S<2>, @L<2>

FIG. 48

STATUS	jpc	jinst	jinstsize	jpcnext	js	b	consume	produce	low	high	NATIVE CODE
(24)	28	iadd	1	29	2	0	2	1	0	2	T28: add S<0>, S<1>
(25)	29	ireturn	1	30	1	0	1	0	0	1	mv r0, S<0> pop lr pop r13 pop r12 pop r11 pop r10 pop r9 pop r8 addi sp, #(nLocal-nArg+nStack)*4 jmp lr

FIG. 49

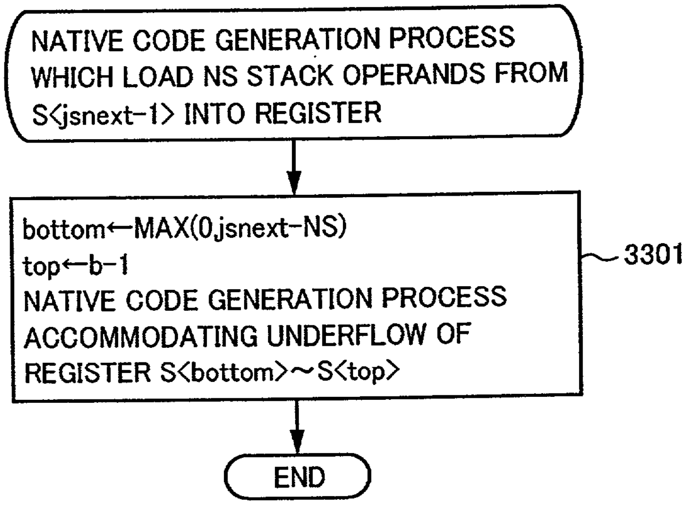

NATIVE CODE GENERATION PROCESS WHICH LOAD NS STACK OPERANDS FROM S<jsnext-1> INTO REGISTER
bottom←MAX(0jsnext-NS)
top←b-1
NATIVE CODE GENERATION PROCESS ACCOMMODATING UNDERFLOW OF REGISTER S<bottom>~S<top>
3301
END

FIG. 50

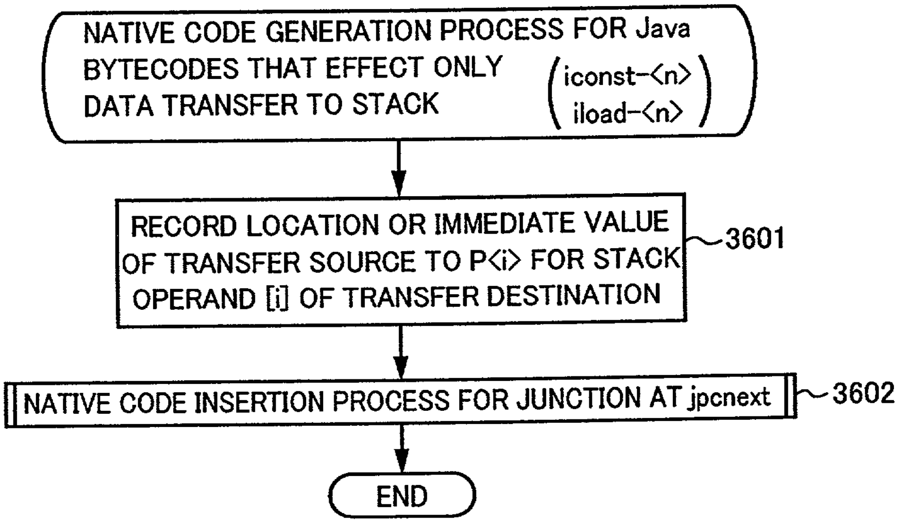

NATIVE CODE GENERATION PROCESS FOR Java BYTECODES THAT EFFECT ONLY DATA TRANSFER TO STACK
(iconst-<n>
iload-<n>)
RECORD LOCATION OR IMMEDIATE VALUE OF TRANSFER SOURCE TO P<i> FOR STACK OPERAND [i] OF TRANSFER DESTINATION
3601
NATIVE CODE INSERTION PROCESS FOR JUNCTION AT jpcnext
3602
END

FIG. 51

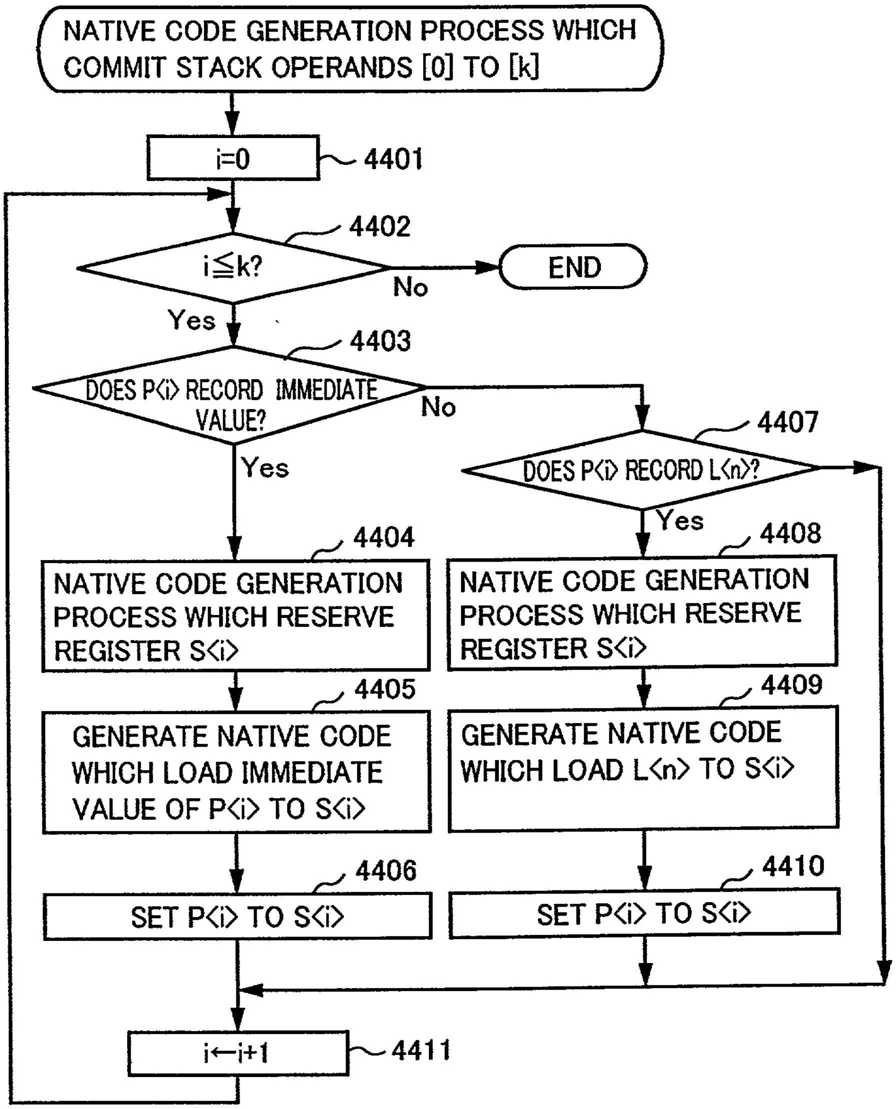

NATIVE CODE GENERATION PROCESS WHICH COMMIT STACK OPERANDS [0] TO [k]
i=0
4401
4402
i≦k?
No
END
Yes
4403
DOES P<i> RECORD IMMEDIATE VALUE?
No
4407
DOES P<i> RECORD L<n>?
Yes
Yes
4404
NATIVE CODE GENERATION PROCESS WHICH RESERVE REGISTER S<i>
4408
NATIVE CODE GENERATION PROCESS WHICH RESERVE REGISTER S<i>
4405
GENERATE NATIVE CODE WHICH LOAD IMMEDIATE VALUE OF P<i> TO S<i>
4409
GENERATE NATIVE CODE WHICH LOAD L<n> TO S<i>
4406
SET P<i> TO S<i>
4410
SET P<i> TO S<i>
i←i+1
4411

FIG. 52

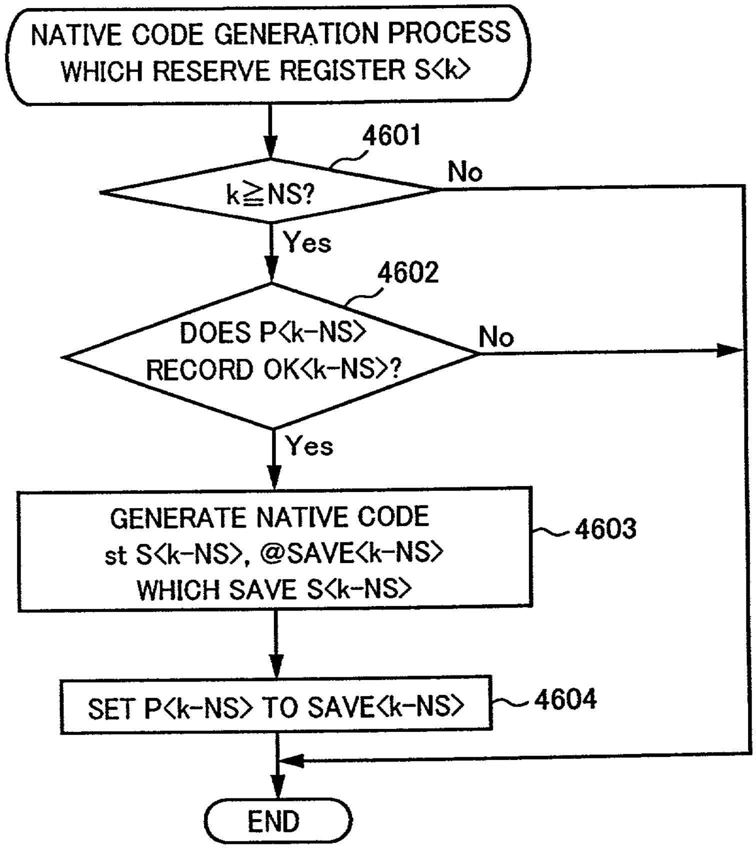

NATIVE CODE GENERATION PROCESS WHICH RESERVE REGISTER S<k>
4601
k≧NS?
No
Yes
4602
DOES P<k-NS> RECORD OK<k-NS>?
No
Yes
GENERATE NATIVE CODE st S<k-NS>, @SAVE<k-NS> WHICH SAVE S<k-NS>
4603
SET P<k-NS> TO SAVE<k-NS>
4604
END

FIG. 53

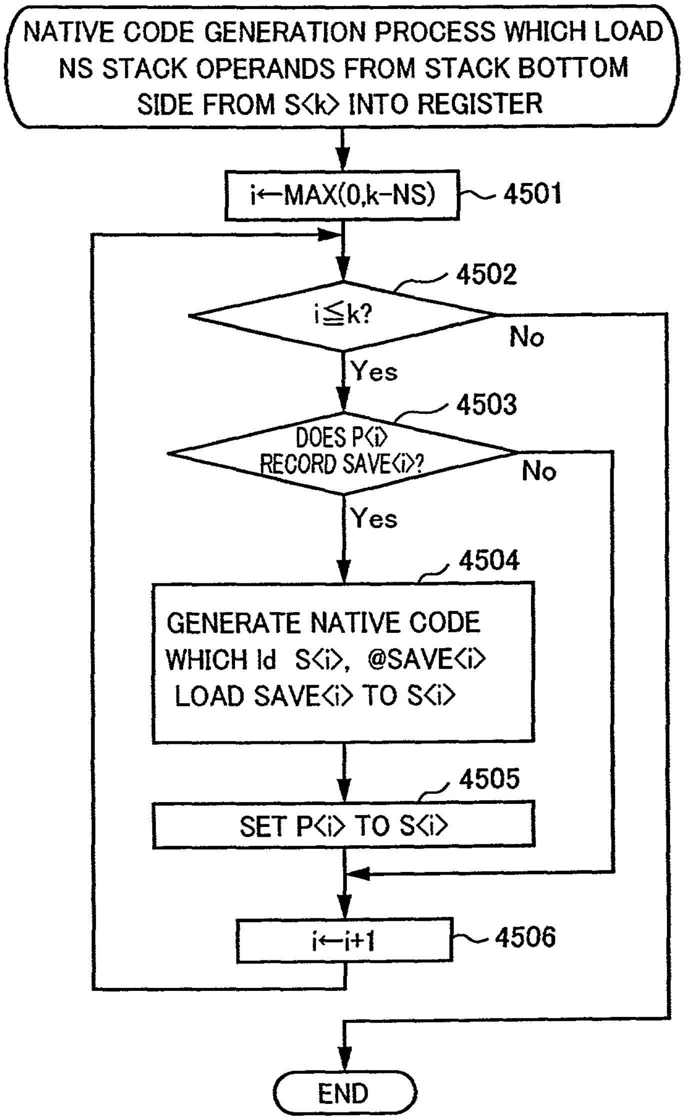

NATIVE CODE GENERATION PROCESS WHICH LOAD NS STACK OPERANDS FROM STACK BOTTOM SIDE FROM S<k> INTO REGISTER
i←MAX(0,k−NS)
4501
4502
i≦k?
No
Yes
4503
DOES P<i> RECORD SAVE<i>?
No
Yes
4504
GENERATE NATIVE CODE WHICH ld S<i>, @SAVE<i> LOAD SAVE<i> TO S<i>
4505
SET P<i> TO S<i>
i←i+1
4506
END

FIG. 54

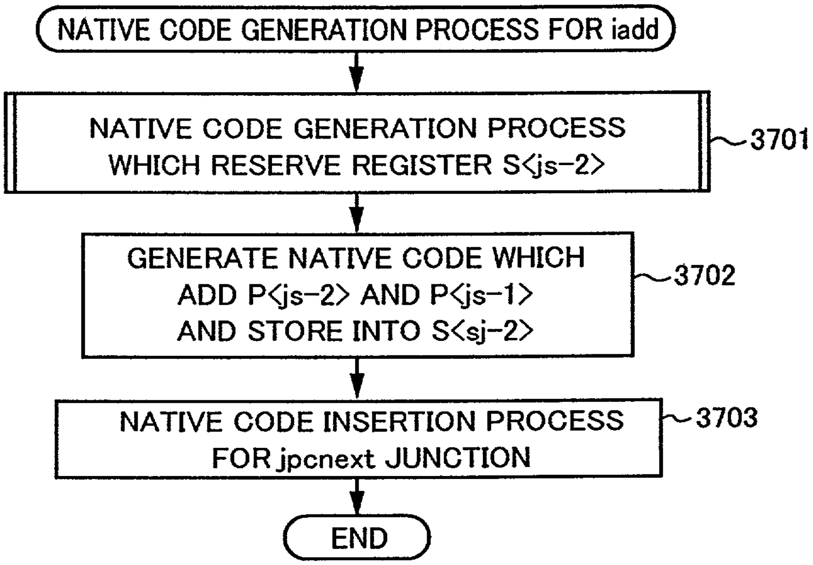

NATIVE CODE GENERATION PROCESS FOR iadd
NATIVE CODE GENERATION PROCESS WHICH RESERVE REGISTER S<js-2>
3701
GENERATE NATIVE CODE WHICH ADD P<js-2> AND P<js-1> AND STORE INTO S<sj-2>
3702
NATIVE CODE INSERTION PROCESS FOR jpcnext JUNCTION
3703
END

FIG. 55

Java bytecode	Operand Allocation		Case No.	Native Code	Other Process
	P<js-2>	P<js-1>			
iadd	Immediate value (below 16-bit signed range)	Immediate value (below 16-bit signed range)	1	None	Record immediate value of P<js-2>+P<js-1> in P<js-2>
		S<js-1>	2	add3 S<js-2>, S<js-1>, #P<js-2>	Record S<js-2> in P<js-2>
		SAVE<js-1>	3	ld r0, @SAVE<js-1> add3 S<js-2>, r0, #P<js-2>	
		Local variable L<n> (register)	4	add3 S<js-2>, L<n>, #P<js-2>	
		Local variable L<n> (memory)	5	ld S<js-2>, @L<n> add3 S<js-2>, S<js-2>, #P<js-2>	
	Immediate value (above 16-bit signed range)	Immedaite value (below 16-bit signed range)	6	None	Record immediate value of P<js-2>+P<js-1>inP<js-2>
		S<js-1>	7	ldh S<js-2>, #high(P<js-2>) or3 S<js-2>, S<js-2>, #low(P<js-2>) add S<js-2>, S<js-1>	RecordS<js-2> in P<js-2>
		SAVE<js-1>	8	ld r0, @SAVE<js-1> ldh S<js-2>, #high(P<js-2>) or3 S<js-2>, S<js-2>, #low(P<js-2>) add S<js-2>, r0	
		Local variable L<n> (register)	9	ldh S<js-2>, #high(P<js-2>) or3 S<js-2>, S<js-2>, #low(P<js-2>) add S<js-2>, L<n>	
		Local variable L<n> (memory)	10	ldh S<js-2>, #high(P<js-2>) or3 S<js-2>, S<js-2>, #low(P<js-2>) ld r0, @L<n> add S<js-2>, r0	
	S<js-2>	Immediate value (below8-bit signed range)	11	addi S<js-2>, #P<js-1>	Record S<js-2> in P<js-2>
		Immediate value (below 16-bit signed range)	12	add3 S<js-2>, S<js-2>, #P<js-1>	
		Immediate value (above 16-bit signed range)	13	ldh r0, #high(P<js-1>) or3 r0, r0, #low(P<js-1>) add S<js-2>, r0	
		S<js-1>	14	add S<js-2>, S<js-1>	
		SAVE<js-1>	15	ld r0, @S<js-1> add S<js-2>, r0	
		Local variable L<n> (register)	16	add S<js-2>, L<n>	
		Local variable L<n> (memory)	17	ld r0, @L<n> add S<js-2>, r0	
	SAVE<js-2>	Immediate value (below 8-bit signed range)	18	ld S<js-2>, @SAVE<js-2> addi S<js-2>, #P<js-1>	Record S<js-2> in P<js-2>
		Immediate value (below 16-bit signed range)	19	ld S<js-2>, @SAVE<js-2> add3 S<js-2>, S<js-2>, #P<js-1>	
		Immediate value (above 16-bit signed range)	20	ld S<js-2>, @SAVE<js-2> ldh r0, #high(P<js-1>) or3 r0, r0, #low(P<js-1>) add S<js-2>, r0	
		S<js-1>	21	ld S<js-2>, @SAVE<js-2> add S<js-2>, S<js-1>	
		SAVE<js-1>	22	ld S<js-2>, @SAVE<js-2> ld r0, @S<js-1> add S<js-2>, r0	
		Local variable L<n> (register)	23	ld S<js-2>, @SAVE<js-2> add S<js-2>, L<n>	
		Local variable L<n> (memory)	24	ld S<js-2>, @SAVE<js-2> ld r0, @L<n> add S<js-2>, r0	
	Local variable L<m> (register)	Immediate value (below 16-bit signed range)	25	add3 S<js-2>, L<n>, #P<js-1>	Record S<js-2> in P<js-2>
		Immediate value (above 16-bit signed range)	26	mv S<js-2>, L<n> ldh r0, #high(P<js-1>) or3 r0, r0, #low(P<js-1>) add S<js-2>, r0	
		S<js-1>	27	mv S<js-2>, L<m> add S<js-2>, S<js-1>	
		SAVE<js-1>	28	ld r0, @SAVE<js-1> mv S<js-2>, L<m> add S<js-2>, r0	
		Local variable L<n> (register)	29	mv S<js-2>, L<m> add S<js-2>, L<n>	
		Local variable L<n> (memory)	30	ld S<js-2>, @L<n> add S<js-2>, L<m>	
	Local variable L<m> (memory)	Immediate value (below 8-bit signed range)	31	ld S<js-2>, @L<n> addi S<js-2>, #P<js-1>	Record S<js-2> in P<js-2>
		Immediate value (below 16-bit signed range)	32	ld S<js-2>, @L<n> add3 S<js-2>, S<js-2>, #P<js-1>	
		Immediate value (above 16-bit signed range)	33	ld S<js-2>, @L<n> ldh r0, #high(P<js-1>) or3 r0, r0, #low(P<js-1>) add S<js-2>, r0	
		S<js-1>	34	ld S<js-2>, @L<m> add S<js-2>, S<js-1>	
		SAVE<js-1>	35	ld r0, @SAVE<js-1> ld S<js-2>, @L<m> add S<js-2>, r0	
		Local variable L<n> (register)	36	ld S<js-2>, @L<m> add S<js-2>, L<n>	
		Local variable L<n> (Memory)	37	ld S<js-2>, @L<m> ld r0,@L<n> add S<js-2>, r0	

FIG. 56

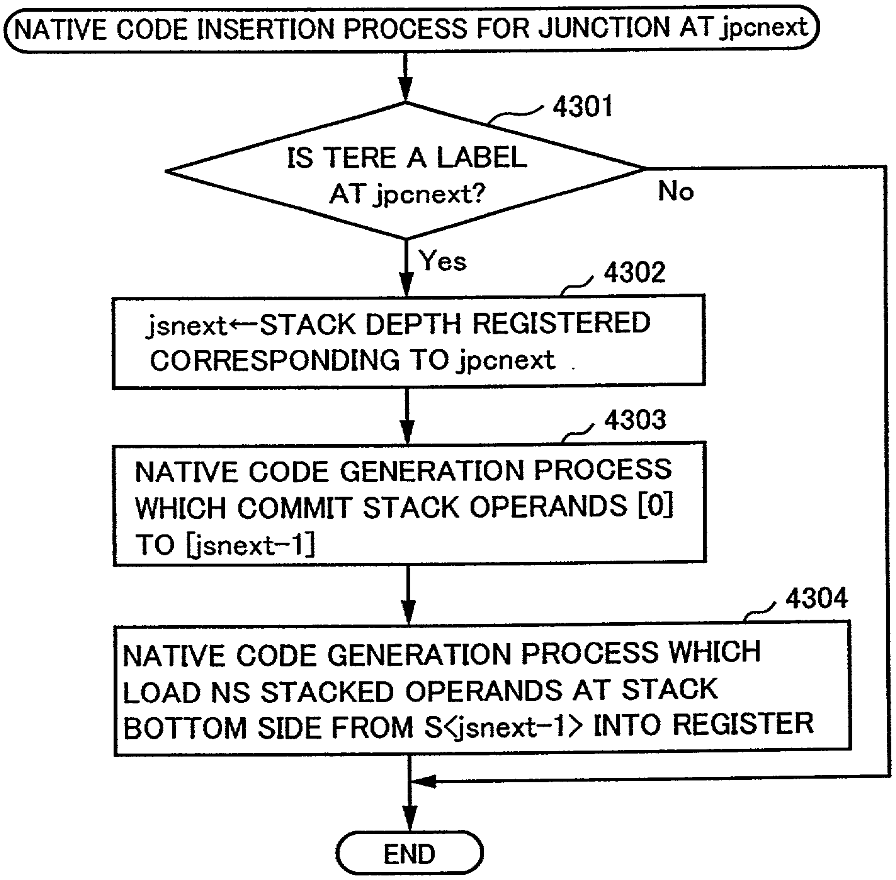

NATIVE CODE INSERTION PROCESS FOR JUNCTION AT jpcnext
4301
IS TERE A LABEL AT jpcnext?
No
Yes
4302
jsnext←STACK DEPTH REGISTERED CORRESPONDING TO jpcnext
4303
NATIVE CODE GENERATION PROCESS WHICH COMMIT STACK OPERANDS [0] TO [jsnext−1]
4304
NATIVE CODE GENERATION PROCESS WHICH LOAD NS STACKED OPERANDS AT STACK BOTTOM SIDE FROM S<jsnext−1> INTO REGISTER
END

FIG. 57

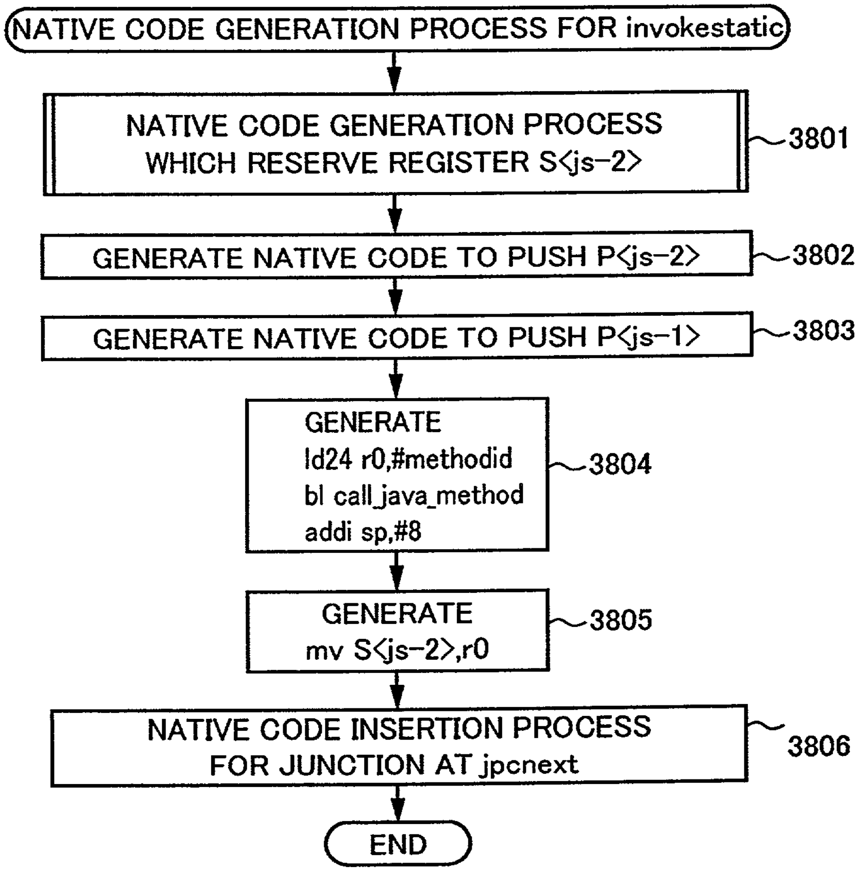

NATIVE CODE GENERATION PROCESS FOR invokestatic
NATIVE CODE GENERATION PROCESS WHICH RESERVE REGISTER S<js-2>
3801
GENERATE NATIVE CODE TO PUSH P<js-2>
3802
GENERATE NATIVE CODE TO PUSH P<js-1>
3803
GENERATE
ld24 r0,#methodid
bl call_java_method
addi sp,#8
3804
GENERATE
mv S<js-2>,r0
3805
NATIVE CODE INSERTION PROCESS FOR JUNCTION AT jpcnext
3806
END

FIG. 58

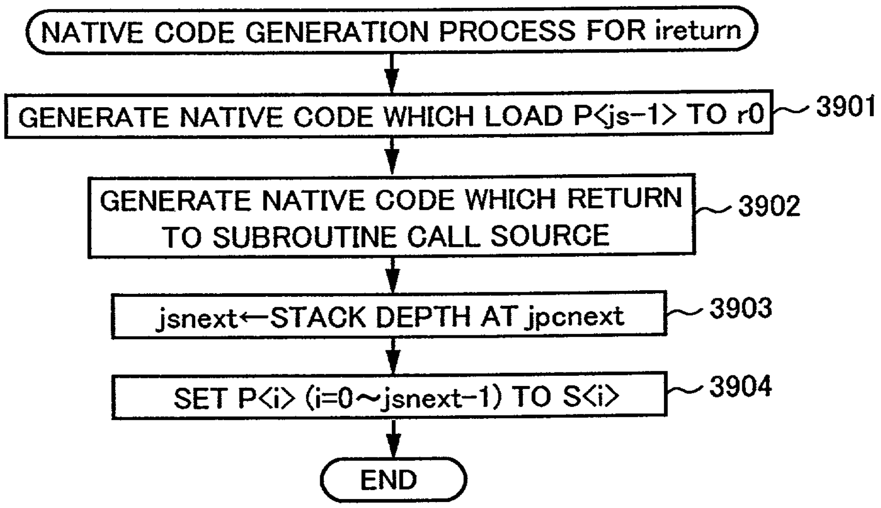

NATIVE CODE GENERATION PROCESS FOR ireturn
GENERATE NATIVE CODE WHICH LOAD P<js-1> TO r0
3901
GENERATE NATIVE CODE WHICH RETURN TO SUBROUTINE CALL SOURCE
3902
jsnext←STACK DEPTH AT jpcnext
3903
SET P<i> (i=0~jsnext-1) TO S<i>
3904
END

FIG. 59

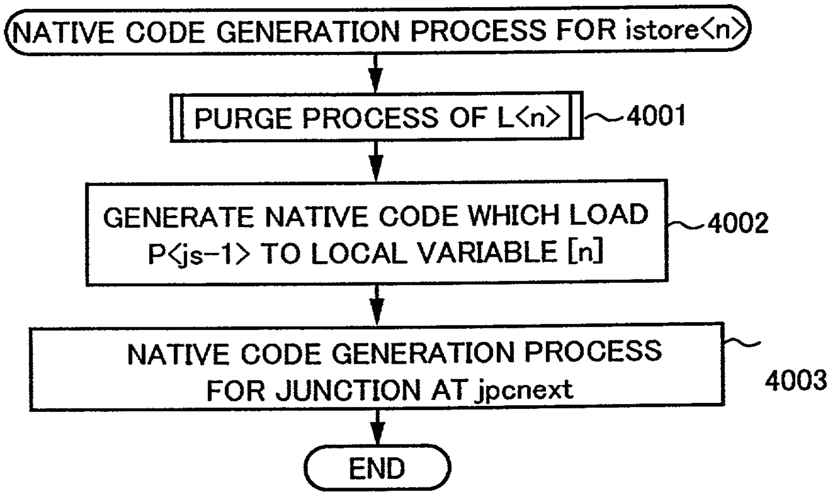

NATIVE CODE GENERATION PROCESS FOR istore<n>
PURGE PROCESS OF L<n>
4001
GENERATE NATIVE CODE WHICH LOAD P<js-1> TO LOCAL VARIABLE [n]
4002
NATIVE CODE GENERATION PROCESS FOR JUNCTION AT jpcnext
4003
END

FIG. 60

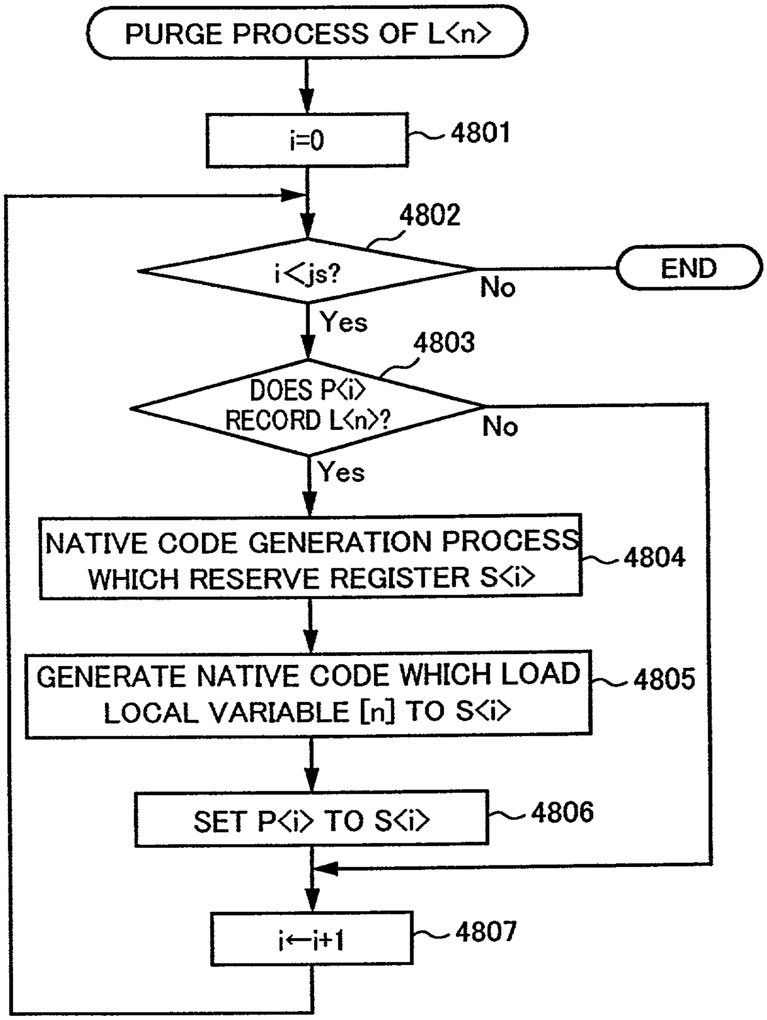

PURGE PROCESS OF L<n>
i=0
4801
4802
i<js?
No
END
Yes
4803
DOES P<i> RECORD L<n>?
No
Yes
NATIVE CODE GENERATION PROCESS WHICH RESERVE REGISTER S<i>
4804
GENERATE NATIVE CODE WHICH LOAD LOCAL VARIABLE [n] TO S<i>
4805
SET P<i> TO S<i>
4806
i←i+1
4807

FIG. 61

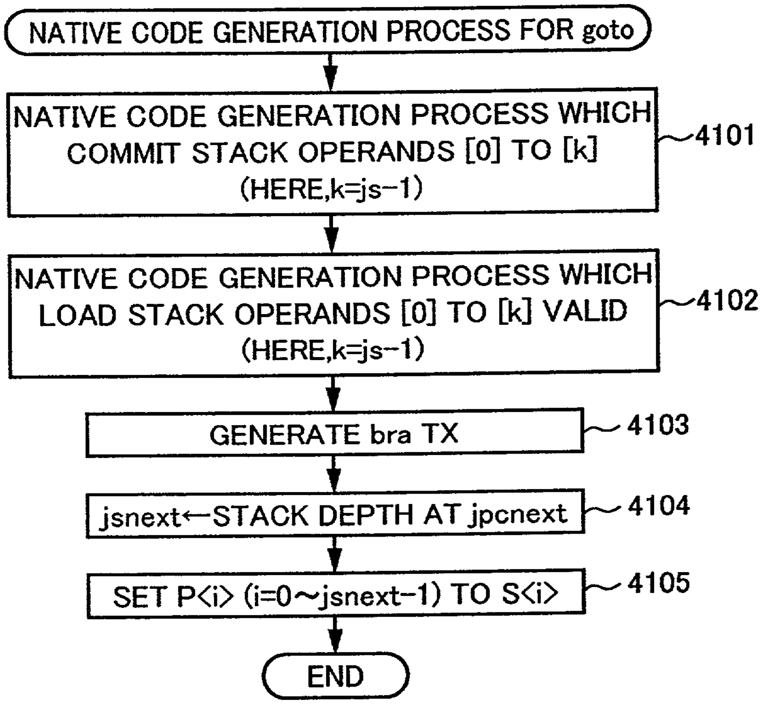

NATIVE CODE GENERATION PROCESS FOR goto
NATIVE CODE GENERATION PROCESS WHICH COMMIT STACK OPERANDS [0] TO [k] (HERE,k=js−1)
4101
NATIVE CODE GENERATION PROCESS WHICH LOAD STACK OPERANDS [0] TO [k] VALID (HERE,k=js−1)
4102
GENERATE bra TX
4103
jsnext←STACK DEPTH AT jpcnext
4104
SET P<i> (i=0~jsnext−1) TO S<i>
4105
END

FIG. 62

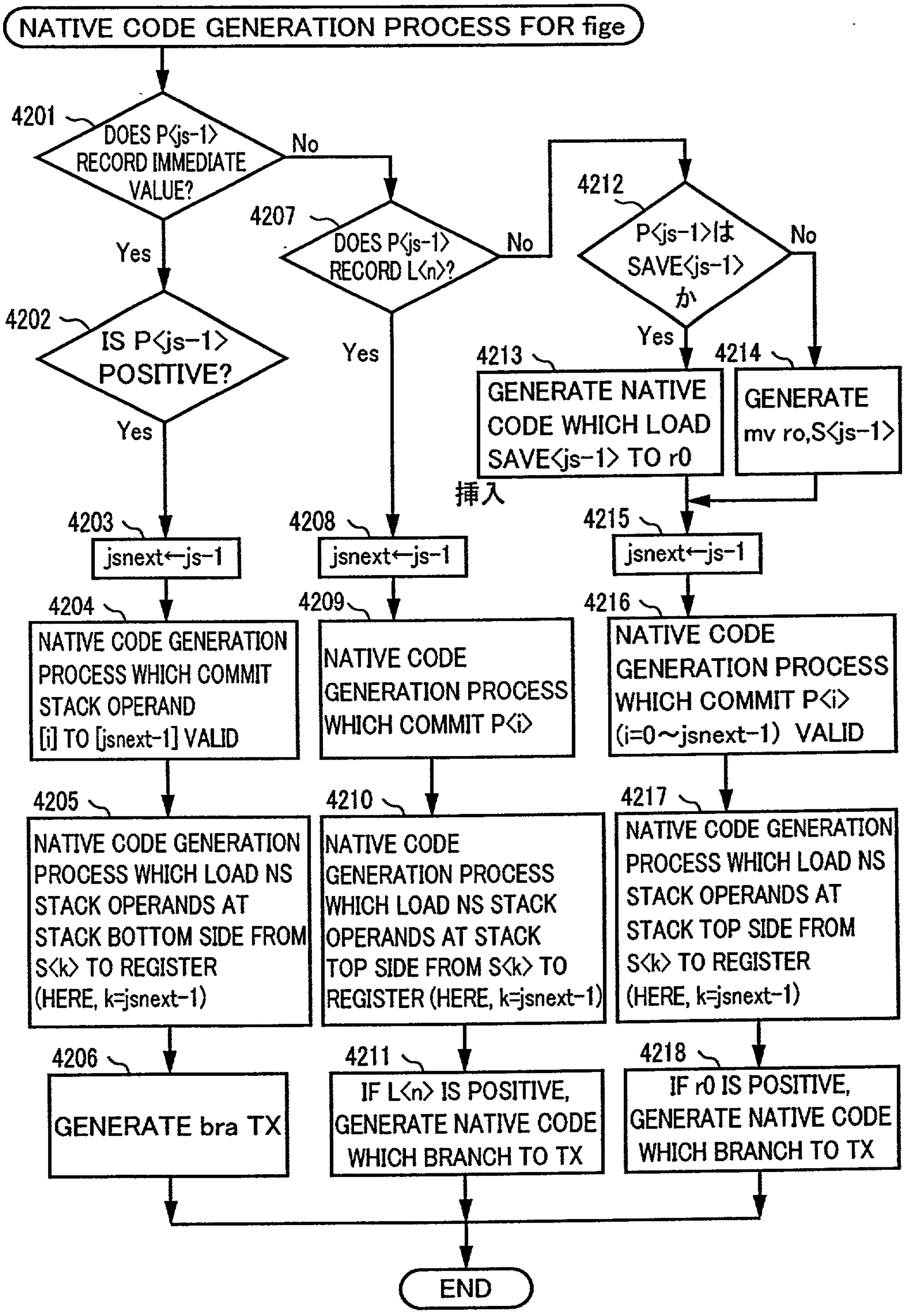

NATIVE CODE GENERATION PROCESS FOR fige
4201
DOES P<js-1> RECORD IMMEDIATE VALUE?
No
Yes
4202
IS P<js-1> POSITIVE?
Yes
4203
jsnext←js-1
4204
NATIVE CODE GENERATION PROCESS WHICH COMMIT STACK OPERAND [i] TO [jsnext-1] VALID
4205
NATIVE CODE GENERATION PROCESS WHICH LOAD NS STACK OPERANDS AT STACK BOTTOM SIDE FROM S<k> TO REGISTER (HERE, k=jsnext-1)
4206
GENERATE bra TX
4207
DOES P<js-1> RECORD L<n>?
No
Yes
4208
jsnext←js-1
4209
NATIVE CODE GENERATION PROCESS WHICH COMMIT P<i>
4210
NATIVE CODE GENERATION PROCESS WHICH LOAD NS STACK OPERANDS AT STACK TOP SIDE FROM S<k> TO REGISTER (HERE, k=jsnext-1)
4211
IF L<n> IS POSITIVE, GENERATE NATIVE CODE WHICH BRANCH TO TX
4212
P<js-1>は SAVE<js-1> か
No
Yes
4213
GENERATE NATIVE CODE WHICH LOAD SAVE<js-1> TO r0
挿入
4214
GENERATE mv ro,S<js-1>
4215
jsnext←js-1
4216
NATIVE CODE GENERATION PROCESS WHICH COMMIT P<i> (i=0~jsnext-1) VALID
4217
NATIVE CODE GENERATION PROCESS WHICH LOAD NS STACK OPERANDS AT STACK TOP SIDE FROM S<k> TO REGISTER (HERE, k=jsnext-1)
4218
IF r0 IS POSITIVE, GENERATE NATIVE CODE WHICH BRANCH TO TX
END

FIG. 63

STATUS	jpc	jinst	jinstsize	jpcnext	js	P<0>	P<1>	P<2>	P<3>	P<4>	P<5>	NATIVE CODE
(1)						—	—	—	—	—	—	addi sp, #-(nLocal-nArg+nStack)*4 push r8 push r9 push r10 push r11 push r12 push r13 push lr ld L<0>, @((nLocal+nStack+nSave-1)*4,sp) ld L<1>, @((nLocal+nStack+nSave-2)*4,sp)
(2)	0	iload_0	1	1	0	L<0>	—	—	—	—	—	
(3)	1	iload_1	1	2	1	L<0>	L<1>	—	—	—	—	
(4)	2	iadd	1	3	2	S<0>	—	—	—	—	—	mv S<0>, L<0> add S<0>, L<1>
(5)	3	istore_2	1	4	1	—	—	—	—	—	—	st S<0>, @L<2>
(6)	4	iconst_1	1	5	0	1	—	—	—	—	—	
(7)	5	iload_0	1	6	1	1	L<0>	—	—	—	—	
(8)	6	ifge 21	3	9	2	S<0>	—	—	—	—	—	ldi S<0>, #1 bgez L<0>, T17
(9)	9	iconst_2	1	10	1	S<0>	2	—	—	—	—	
(10)	10	iload_0	1	11	2	S<0>	2	L<0>	—	—	—	
(11)	11	iload_1	1	12	3	S<0>	2	L<0>	L<1>	—	—	
(12)	12	iconst_3	1	13	4	S<0>	2	L<0>	L<1>	3	—	
(13)	13	iload_2	1	14	5	S<0>	2	L<0>	L<1>	3	L<2>	
(14)	14	iadd	1	15	6	SAVE<0>	2	L<0>	L<1>	S<4>	—	st S<0>, @SAVE<0> ld S<4>, @L<2> add3 S<4>, S<4>, #3
(15)	15	idiv	1	16	5	SAVE<0>	2	L<0>	S<3>	—	—	mv S<3>, L<1> div S<3>, S<4>
(16)	16	iadd	1	17	4	SAVE<0>	2	S<2>	—	—	—	mv S<2>, L<0> add S<2>, S<3>
(17)	17	imul	1	18	3	SAVE<0>	S<1>	—	—	—	—	sll3 S<1>, S<2>, #1
(18)	18	goto 28	3	21	2	S<0>	S<1>	—	—	—	—	ld S<0>, @SAVE<0> bra T28
(18')						S<0>	—	—	—	—	—	

FIG. 64

STATUS	jpc	jinst	jinstsize	jpcnext	js	P<0>	P<1>	P<2>	P<3>	P<4>	P<5>	NATIVE CODE
(19)	21	iload_0	1	22	1	S<0>	L<0>	—	—	—	—	21:
(20)	22	iconst_1	1	23	2	S<0>	L<0>	1	—	—	—	
(21)	23	isub	1	24	3	S<0>	S<1>	—	—	—	—	add3 S<1>, L<0>, #-1
(22)	24	iload_2	1	25	2	S<0>	S<1>	L<2>	—	—	—	
(23)	25	invokestatic <int F(int, int)>	3	28	3	S<0>	S<1>	—	—	—	—	ld r0, @L<2> push r0 push S<1> ld24 r0, #methodId jl callJavaMethod addi sp, #8 mv S<1>, r0
(24)	28	iadd	1	29	2	S<0>	—	—	—	—	—	28: add S<0>, S<1>
(25)	29	ireturn	1	30	1	—	—	—	—	—	—	mv r0, S<0> pop lr pop r13 pop r12 pop r11 pop r10 pop r9 pop r8 addi sp, #(nLocal-nArg+nStack)*4 jmp lr